UC Riverside
UC Riverside Electronic Theses and Dissertations

Title
Heterogeneous Byzantine Replicated Systems

Permalink
https://escholarship.org/uc/item/2kt7w4nb

Author
Li, Xiao

Publication Date
2024

Copyright Information
This work is made available under the terms of a Creative Commons Attribution-NonCommercial-ShareAlike License, available at https://creativecommons.org/licenses/by-nc-sa/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
RIVERSIDE

Heterogeneous Byzantine Replicated Systems

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Xiao Li

September 2024

Dissertation Committee:

Dr. Mohsen Lesani, Chairperson
Dr. Philip Brisk
Dr. Vassilis Tsotras
Dr. Heng Yin

Copyright by
Xiao Li
2024

The Dissertation of Xiao Li is approved:

Committee Chairperson

University of California, Riverside

Acknowledgments

It has been a long journey for me to reach the end of the Ph.D program. I joined University of California, Riverside as a Master student and later transferred to the Ph.D program. At the S3 lab, I spent valuable times of my life to explore different research problems, design protocols, write proofs, build prototype systems and present our works. I would like to thank all the people who have helped and supported me along this way, which made it possible for me to achieve so many accomplishments. First, I would like to thank my advisor, who guided me to the world of formal methods and distributed system research. He invited me to work in his S3 lab when I was a Master student and together we published my first paper [292]. From then on, he has been providing countless help every step of the way. I would not be writing this PhD dissertation without his exceptional guidance. I also would like to thank my family for all the encouragement and support: my father Li Yi, my mother, Wang Mei, my grandmother Hu Ling and my grandfather, Li Jingze. They have always been proud of me and hold beliefs in my potentials. I would like to thank my partner and closest friend, Zhenxiao Qi, whose love and support are priceless. We met each other at the beginning of my Ph.D study and since then he has been there for me no matter happy or sad, good or bad. We are the pillars of each other in this challenging journey and we both made it! I also want to thank my friends and the members of UCR's karate club including but not limited to Sensi Amy and Senpai Tobby. Friendship and Karate has been my support network when I stuck on research projects. Last but not least, I want to express my deepest love and appreciation to my cat Boba and dog Ori. Their love are anchors of my life and I own them a lot for being my sweetest support.

Chapter 2 was previously published as “Hamraz: Resilient partitioning and replication” in 2022 IEEE Symposium on Security and Privacy (SP) [292]. The reviewers found “impressed by the rigor of the paper; every concept and process is formalized, and the system provides formal guarantees of noninterference and resilience.” I would like to express my appreciation to both my advisor and the second author Dr. Farzin Houshmand. My advisor provided valuable support from formalizing the ideas to analyzing the experiment results. Farzin made great contributions for the experiment section.

Chapter 3 was previously published as “Quorum Subsumption for Heterogeneous Quorum Systems” in DISC ’23 (The International Symposium on Distributed Computing) [289]. I would like to thank both the second author Eric and Dr. Giuliano Losa from Stellar Development Foundation. Without their help, it is not possible to complete my proudest yet challenging project. I would like to express special appreciation to Dr.Guliano Losa. He spent a lot of time to discuss with me the subtle details of the concrete consensus protocol, which content was later invited to the journal submission for DISC Special Issue. I also want to thank Prof. Rotem Oshman, the editor of DISC 2023 for recognizing the fundamental importance of our paper and inviting us to submit an extended version to the DISC 2023 Special Issue.

Chapter 4 was recently accepted as “Brief Announcement: Reconfigurable Heterogeneous Quorum Systems” in DISC’ 24 (The International Symposium on Distributed Computing). The full version is available on arXiv [295]. I would like to thank my advisor for all the insightful discussions.

Chapter 5 is an ongoing collaboration with my colleague Tejas. I would like to thank him for implementing all the protocols I designed. The experiment results are important contributions for this paper and will definitely make it a stronger submission.

Finally, I would like to express my appreciation to all the grants, fundings, fellowships and awards for their generous support: NSF CAREER: Distributed System Synthesis on Certified Middleware (Award Number: 1942711); DARPA Information and Vulnerability Flow Type Systems (Grant number: D22AP00146-00); NSF CRII: SHF: Certified Byzantine Fault-tolerant Systems (Award Number: 1657204); Dissertation Year Completion Fellowhip (UC Riverside); Grace Hopper Conference Scholarship 2023 (UC Riverside); Dissertation Year Program Fellowhip 2023/2024 (UC Riverside); Student Travel Grant for 2022 ACM Conference on Computer and Communications Security (CCS); GSA Travel Award (UC Riverside); Student Travel Award, 2022 IEEE Symposium on Security and Privacy (S&P).

To *Ori* and *Boba* for being the stars of my life.

ABSTRACT OF THE DISSERTATION

Heterogeneous Byzantine Replicated Systems

by

Xiao Li

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, September 2024
Dr. Mohsen Lesani, Chairperson

Since the advent of Byzantine replicated systems such as PBFT, numerous follow-up variants can maintain consistent replications in the presence of malicious nodes. However, Byzantine replication systems have been traditionally monolithic and homogeneous: the quorums nodes trust are fixed and uniform. This dissertation considers multiple aspects of heterogeneity for Byzantine replicated systems.

First, this dissertation considers heterogeneous replicated systems that ensure the trustworthiness specifications for a given computation. In the face of malicious Byzantine attacks, the ultimate goal is to assure end-to-end policies for confidentiality, integrity and availability. This dissertation presents a security-typed object-based language, a partitioning transformation to partition methods, an operational semantics and an information flow type inference system for replicated classes, and a synthesis tool called Hamraz. Given a class and the specification of its end-to-end policies, Hamraz applies type inference to automatically place and replicate the fields and methods of the class on Byzantine quorum systems, and synthesizes trustworthy-by-construction distributed systems. The experiments

show the resiliency of the resulting systems; they can gracefully tolerate attacks that are as strong as the specified policies.

Second, this dissertation investigates heterogeneous quorum systems (HQS) where each process can declare its own quorums. In traditional replication systems, the set of trusted quorums is homogeneous across processes. However, trust is a subjective matter. This dissertation presents a general model of HQS. When quorum systems are not uniform, the properties that they should maintain to support reliable broadcast and consensus are not well understood. It was shown that quorum intersection and availability are necessary. This dissertation proves that they are not sufficient. It then defines the notion of quorum subsumption, and shows that the three conditions together are sufficient: it presents reliable broadcast and consensus protocols and proves their correctness.

Thirdly, this dissertation investigate reconfiguration protocols for HQS. Traditionally, the trusted set of quorums was not only homogeneous but fixed. However, trust evolves; processes might need to change their quorums. This dissertation presents reconfiguration protocols for HQS including joining and leaving of a process, and adding and removing of a quorum, and further, proves their correctness. The design of the protocols is informed by the trade-offs for the properties that reconfigurations can preserve. The dissertation further presents a graph characterization of HQS and its application for reconfiguration optimization.

Finally, this dissertation considers heterogeneous clustered replication where nodes are divided into clusters based on proximity to enable scalability across the globe. Existing clustered replication protocols are homogeneous and closed: the number of nodes across

clusters is the same and fixed. This dissertation presents heterogeneous and reconfigurable clustered replication. It presents a clustered replication protocol that supports different cluster sizes, and further, allows processes to join and leave clusters. It formally states and proves the safety and liveness properties of proposed protocols.

Contents

List of Figures xiv

List of Tables xvi

1 Introduction 1
- 1.1 Overview of Contributions . . . 2
  - 1.1.1 Resilient Partitioning and Replication . . . 2
  - 1.1.2 Quorum Subsumption for Heterogeneous Quorum Systems . . . 3
  - 1.1.3 Reconfigurable Heterogeneous Quorum Systems . . . 4
  - 1.1.4 Reconfigurable Clustered Byzantine Replication . . . 4

2 Resilient Partitioning and Replication 6
- 2.1 Introduction . . . 6
- 2.2 Overview . . . 11
- 2.3 Classes and Security Types . . . 18
- 2.4 Partitioning . . . 23
- 2.5 Operational Semantics . . . 28
- 2.6 Information Flow Type System . . . 33
- 2.7 Security and Resiliency Guarantees . . . 38
- 2.8 Constraint Solving . . . 44
- 2.9 Implementation and Experiments . . . 45
- 2.10 Related Works . . . 49
- 2.11 Conclusion . . . 51
- 2.12 Constraint Solving . . . 53
- 2.13 Security Guarantees . . . 59
  - 2.13.1 Confidentiality Non-Interference . . . 59
  - 2.13.2 Integrity Non-Interference . . . 63
  - 2.13.3 Availability Non-Interference . . . 66
  - 2.13.4 Integrity Resilience . . . 68
  - 2.13.5 Availability Resilience . . . 73
  - 2.13.6 Helper Lemmas . . . 76

3 Quorum Subsumption for Heterogeneous Quorum Systems 80
3.1 Introduction 80
3.2 Heterogeneous Quorum Systems 83
3.2.1 Processes and Quorums 84
3.2.2 Properties 85
3.3 Protocol Implementation 92
3.4 Protocol Specification 94
3.5 Impossibility 95
3.5.1 Consensus 97
3.5.2 Byzantine Reliable Broadcast 100
3.6 Protocols 101
3.6.1 Reliable Broadcast Protocol 103
3.6.2 Byzantine Consensus Protocol 110
3.6.3 Practical Byzantine Consensus Protocol 119
3.7 Example Execution for Consensus 125
3.8 Discussion 128
3.9 Related Works 129
3.10 Conclusion 133

4 Reconfigurable Heterogeneous Quorum Systems 134
4.1 Introduction 134
4.2 Quorum Systems 138
4.3 Properties 141
4.4 Graph Characterization 147
4.5 Reconfiguration and Trade-offs 151
4.6 Leave and Remove 156
4.7 Add 162
4.8 Sink Discovery 165
4.9 Join 169
4.10 AC Leave and Remove 171
4.10.1 Correctness 171
4.11 Add 173
4.11.1 Protocol 179
4.11.2 Correctness 184
4.12 PC Leave and Remove 188
4.13 Sink Discovery 189
4.14 AC Leave and Remove Proofs 193
4.14.1 Remove, Inclusion-preservation 193
4.14.2 Remove, Availability-preservation 194
4.14.3 Remove, Intersection-preservation 195
4.15 Add Proofs 197
4.15.1 Add, Inclusion-preservation 197
4.15.2 Add, Availability-preservation 199
4.15.3 Add, Intersection-preservation 200
4.16 Sink Discovery Proofs 202

4.16.1 Sink Discovery, Completeness 202
4.16.2 Sink Discovery, Accuracy 203
4.17 Discussion 205
4.18 Related Works 206
4.19 Conclusion 210

5 Reconfigurable Clustered Byzantine Replication 211
5.1 Introduction 211
5.2 Overview 215
5.3 Inter-cluster Communication 225
5.4 Reconfiguration 231
5.5 Protocol Phases 240
5.5.1 Correctness 242
5.6 Related Work 245
5.7 Conclusion 248
5.8 Protocol Phases 249
5.9 Proofs 257
5.9.1 Remote Leader Change 257
5.9.2 Inter-cluster Broadcast 262
5.9.3 Byzantine Reliable Dissemination 263
5.9.4 Reconfiguration 266
5.9.5 Replication System 270

6 Conclusions 273

Bibliography 276

List of Figures

2.1 One-Time Transfer. (a) User specification, (b) Partitioned class, (c) Typing, (d) Placement, (e) Execution. $A = \{p_{A1}, .., p_{A7}\}$, $B = \{p_{B1}, .., p_{B4}\}$. The set $\mathcal{P}_n(S)$ is the set of subsets of S of cardinality n. The set $S_{i..j}$ denotes $\{p_{Si}, .., p_{Sj}\}$. 10

2.2 Syntax. $\mathcal{P}(S) = 2^S$ is the power set of S. 19

2.3 Lattices for (a) Confidentiality (b) Integrity and (c) Availability. Arrows show the correct flow direction: more confidentiality, less integrity, and less availability. 19

2.4 Factoring. (a) Factored expressions. (b) Example factored expression $[\![\textit{transfer}\ (x)]\!]\ \textit{res}$. (c) CPS transformation. 24

2.5 Splitting Transformation 26

2.6 Runtime State 30

2.7 Sequential Operational Semantics 30

2.8 Distributed Operational Semantics. It is parametric in terms of the class $C = \langle \overline{o}, \overline{d} \rangle$, the method and object placements $\mathcal{M}$ and $\mathcal{O}$, and the Byzantine principals $\mathcal{B}$. In the ThisCall rule, the union operator $\cup$ is extended for $\bot$ values: $\bot \cup s = \bot$. 31

2.9 Information Flow Type Inference System 34

2.10 Top row: Response time for increased faults. Bottom row: Response time for increased resiliency. $\mathcal{P}_i(A) \times_\cup .. \times_\cup \mathcal{P}_k(Z)$ is denoted as $\langle i, .., k \rangle @ n$ where n is the total number of principals. 46

2.11 Response time for increased load 50

3.1 Quorum System Example 87

3.2 Indistinguishable Executions 99

3.3 Last Minute Attack. $b = \langle 1, 4 \rangle$. The *candidate* of well-behaved leader l_2 is $b' = \langle 2, 3 \rangle$. The votes commit and abort are abbreviated as $\mathbb{C}$ and $\mathbb{A}$. The new leader events are triggered at the black dots at each process. Prepared ballots are shown below the time line for each process. 113

4.1 Example Quorum System 139

4.2 Quorum inclusion of q for P. Process p is a member of q that falls inside P, and q' is a quorum of p. Well-behaved processes of q' (shown as green) should be a subset of q. 144

4.3 Quorum Graph Example 148

4.4 Example Quorum Systems for Trade-offs 153

4.5 The Leave and Remove Protocols, Preserving Quorum Intersection. 158

4.6 The Add protocol, Preserving Quorum Intersection. 162

4.7 The Add protocol, Preserving Quorum Intersection. 174

4.8 Phase 2: Intersection Check, and Phase 3: Update 177

4.9 Phase1: Inclusion Check 180

5.1 Overview of Phases and Sub-protocols 216

5.2 Phase 2: Inter-cluster Communication. $f_1 = 1$, $f_2 = 2$. 218

5.3 Phase 1: Reconfiguration 221

List of Tables

2.1 Partitioning and Type Inference. PT: Partitioning time, CN: Constraints number, GT: Constraint generation time, ST: Constraint solving time, TT: Total time 47

3.1 Non-termination for Bracha protocol with blocking sets 89

3.2 An execution for consensus protocol with leader switch 127

Chapter 1

Introduction

Since the advent of Practical Byzantine Fault Tolerance (PBFT), numerous variants have been developed to maintain consistent replication in the presence of malicious nodes. These protocols are energy-efficient and ensure that all nodes possess equal power within the network, making them an appealing technology for a wide range of applications, including finance, supply chain, and healthcare. Despite their widespread adoption, traditional Byzantine Fault Tolerant (BFT) systems remain monolithic and homogeneous. Typically, nodes from geographically distant regions are grouped together, and the quorums they trust are fixed and uniform. In contrast, Bitcoin and other resource-based blockchains are naturally open and decentralized. However, they suffer from high energy consumption, low throughput, and probabilistic liveness guarantees. This dissertation aims to combine the best of the two worlds: enabling the heterogeneity and openness of Byzantine replicated systems.

1.1 Overview of Contributions

This dissertation explores multiple dimensions of heterogeneity and openness within BFT replicated systems. More specifically, this dissertation presents the heterogeneity of resilient partitioning and replication for BFT systems, formal definitions and conditions for reliable broadcast and consensus in heterogeneous quorum systems (HQS), reconfiguration protocols for HQS and reconfigurable clustered Byzantine replication.

1.1.1 Resilient Partitioning and Replication

With the rise of inter-organization corporations in healthcare, finance, and military, multiple parties with different trust assumptions need to coordinate for a common goal. The heterogeneity of trust assumptions (confidentiality policies) leads to the distribution of data and computation across administrative boundaries, which prohibit simple full replication of the whole system. Moreover, the various integrity and availability policies from different parties result in different levels of replication, which makes system designs error-prone and hard to enforce system-wide policies.

Research about end-to-end security and Byzantine fault tolerance systems have been two separate paths: Information flow control can guarantee that sensitive data does not leak and computed results are correct. But there is no practical abstraction for availability; Byzantine quorum replication protocols can enforce availability in the face of Byzantine failures. However, they are monolithic systems with uniform trust assumptions. Therefore, chapter 2 explores the ultimate goal of assuring end-to-end policies for the three aspects of trustworthiness: confidentiality, integrity and availability in the face of malicious Byzantine

attacks, We present a security-typed object-based language, a partitioning transformation, an operational semantics, and an information flow type inference system for partitioned and replicated classes. The type system provably guarantees that well-typed methods enjoy noninterference for the three properties, and that their types quantify their resilience to Byzantine attacks. Given a class and the specification of its end-to-end policies, our developed tool Hamraz applies type inference to automatically place and replicate the fields and methods of the class on Byzantine quorum systems, and synthesize trustworthy-by-construction distributed systems. The experiments show the resiliency of the resulting systems: they can gracefully tolerate attacks that are as strong as the specified policies.

1.1.2 Quorum Subsumption for Heterogeneous Quorum Systems

The traditional Byzantine quorum-system model assumes a pre-existing, global agreement on the set of quorums (typically defined as the sets consisting of more than two-thirds of the participants). This assumption is problematic in permissionless systems, which strive to allow anyone to join or leave the system dynamically. While proof-of-stake permissionless systems like Ethereum require newly joining participants to register into the system, other permissionless systems like the XRP Ledger or the Stellar network allow participants to join the system without synchronization by forgoing agreement on the set of quorums. This results in what we call a heterogeneous quorum system, where each participant has its own, personal set of quorums.

An important question is to determine under what condition is it possible to solve synchronization problems like reliable broadcast or consensus in a heterogeneous quorum system. In chapter 3, we show that the traditional quorum intersection and quorum avail-

ability conditions are not sufficient in heterogeneous quorum systems. Moreover, we propose quorum subsumption, a new condition which, together with quorum availability and quorum intersection, is sufficient to allow solving reliable broadcast and consensus.

Finally, we propose algorithms for reliable broadcast and consensus in heterogeneous quorum systems that satisfy quorum subsumption.

1.1.3 Reconfigurable Heterogeneous Quorum Systems

In contrast to proof-of-work replication, Byzantine quorum systems maintain consistency across replicas with higher throughput, modest energy consumption, and deterministic liveness guarantees. If complemented with heterogeneous trust and open membership, they have the potential to serve as blockchains backbone. In chapter 4, we present a general model of heterogeneous quorum systems where each participant can declare its own quorums, and captures the consistency, availability and inclusion properties of these systems. In order to support open membership, we then present reconfiguration protocols for heterogeneous quorum systems including joining and leaving of a process, and adding and removing of a quorum, and further, prove their correctness in the face of Byzantine attacks. The design of the protocols is informed by the trade-offs we prove for the properties that reconfigurations can preserve. We further present a graph characterization of heterogeneous quorum systems, and its application for reconfiguration optimization.

1.1.4 Reconfigurable Clustered Byzantine Replication

In order to scale Byzantine replicated systems across the globe, clustered replication protocols divide nodes into clusters based on proximity. However, existing protocols

are homogeneous and closed: the number of nodes across clusters is the same and fixed. Chapter 5 presents heterogeneous and reconfigurable clustered replication. We present a clustered replication protocols that supports different cluster sizes, and further, allows processes to join and leave clusters. We formally states and proves the safety and liveness properties of the protocol.

Chapter 2

Resilient Partitioning and Replication

2.1 Introduction

Building trustworthy systems has been the holy grail of computing. The three desired properties, *confidentiality, integrity, and availability*, also known as the CIA triad, guarantee that sensitive data does not leak, computed results are correct, and the system remains accessible in the face of failures and attacks. Assurance of these properties is particularly needed when multiple principals with partial mutual trust cooperate. Inter-organizational systems are common: business-to-business procurement systems, medical information systems that integrate care-provider institutions, and joint military information systems. The distrust between the components of these systems leads to *distribution of data and computation* across administrative boundaries. However, building distributed systems

that are resilient to both benign (crash) and malign (Byzantine) failures is notoriously complicated.

Given an integrated system, the ultimate question is whether it complies with system-wide trustworthiness policies. Furthermore, given the *end-to-end trustworthiness policies*, how can trustworthy-by-construction systems be automatically constructed? *Information flow control* [147, 393, 374, 426] can enforce end-to-end policies. To preserve confidentiality, it restricts the flow of information from the secret to the public domain. Further, to preserve integrity, a common technique is to compare multiple copies of data or computation against each other, and information flow analysis can check that enough copies are compared [451, 455]. However, this method can reduce availability as all the copies need to be available.

Of the three major aspects of trustworthiness, *availability* has been often dissociated from the others, and unfortunately, sidestepped. Confidentiality and integrity are safety properties but availability is a liveness property. In contrast to safety properties, simply monitoring the system and denying the violating actions cannot provide liveness. A few pioneering works consider information flow control for availability. However, they assume availability of the computation platform [456] or require the user to explicitly program the quorums [457].

Providing availability in the face of Byzantine failures [268] requires sophisticated *Byzantine quorum replication* protocols [326, 114]. A quorum system is a set of quorums such that each is an adequate set of hosts to perform operations. Quorum systems stay available even if only one of their quorums is not compromised. However, Byzantine repli-

cation has been largely regarded as a separate discipline. Further, existing protocols often provide guarantees only for a monolithic system based on assumptions on the Byzantine fraction of the processes. How can Byzantine replication be applied to general computation on integrated systems? Since hosts and polices of an integrated system are often heterogeneous, the deployed quorum systems should vary as the information flows from one computation and storage to another.

This paper *enforces end-to-end policies simultaneously for the three aspects of trustworthiness*, especially the *resiliency of availability*, in the face of *Byzantine attacks*. Further, given the end-to-end policies, it *automatically synthesizes* trustworthy-by-construction distributed systems that guarantee the specified policies. To this end, it presents a *security-typed object-based language*, a *partitioning transformation*, an *operational semantics*, and an *information flow type inference system* for partitioned and replicated classes.

We present a security-typed object-based language to describe classes that can encapsulate multiple field objects to implement their methods. The field objects abstract subsystems and the methods capture their interaction. The language allows the user to specify trustworthiness policies as type annotations. A *security type* consists of three components for the three trustworthiness properties. The space of types for each property is elegantly modeled as a lattice. The confidentiality type of a value is the set of hosts that are trusted to observe or store that value. We represent a failure or attack scenario as a set of principals, i.e., the Byzantine principals. Our novel representation of the *integrity* (and similarly *availability*) *type* of a value is a set of attack scenarios that the integrity (and availability) of the value is *resilient* to.

The language permits a class to be described as a centralized definition with no distribution details. We present a high-level *sequential operational semantics* that model central executions. However, confidentiality types restrict the placement of objects and methods. In particular, a single principal may not be able to host the whole body of a method. Therefore, a method may need to be partitioned and hosted by multiple principals. We present a CPS (continuation-passing style) *transformation to partition* methods. Further, integrity and availability types require *replication* of fields and methods on Byzantine quorum systems. We present a *distributed operational semantics* that model the executions of partitioned and replicated classes. The semantics is parameterized for the placement and replication of the fields and methods.

We present a *type inference system* to enforce policies for the three trustworthiness properties: confidentiality, integrity and availability. It performs dependency analysis and rejects classes that violate the type specifications. In particular, if a method depends on results from another method, then the former can be at most as available as the latter. The type system provably guarantees that *well-typed methods have noninterference* for the three properties. For example, an expression does not access objects of higher confidentiality, lower integrity, or lower availability than its type. Further, *well-typed methods do not go wrong: they are resilient against Byzantine attacks that are as strong as their types.* In particular, if the Byzantine attack is no stronger than the integrity and availability type of a method, then any distributed execution of the method matches its sequential execution.

Given the placement and replication of field objects and methods, the type system can check their adequacy for the type specifications. More importantly, given the type

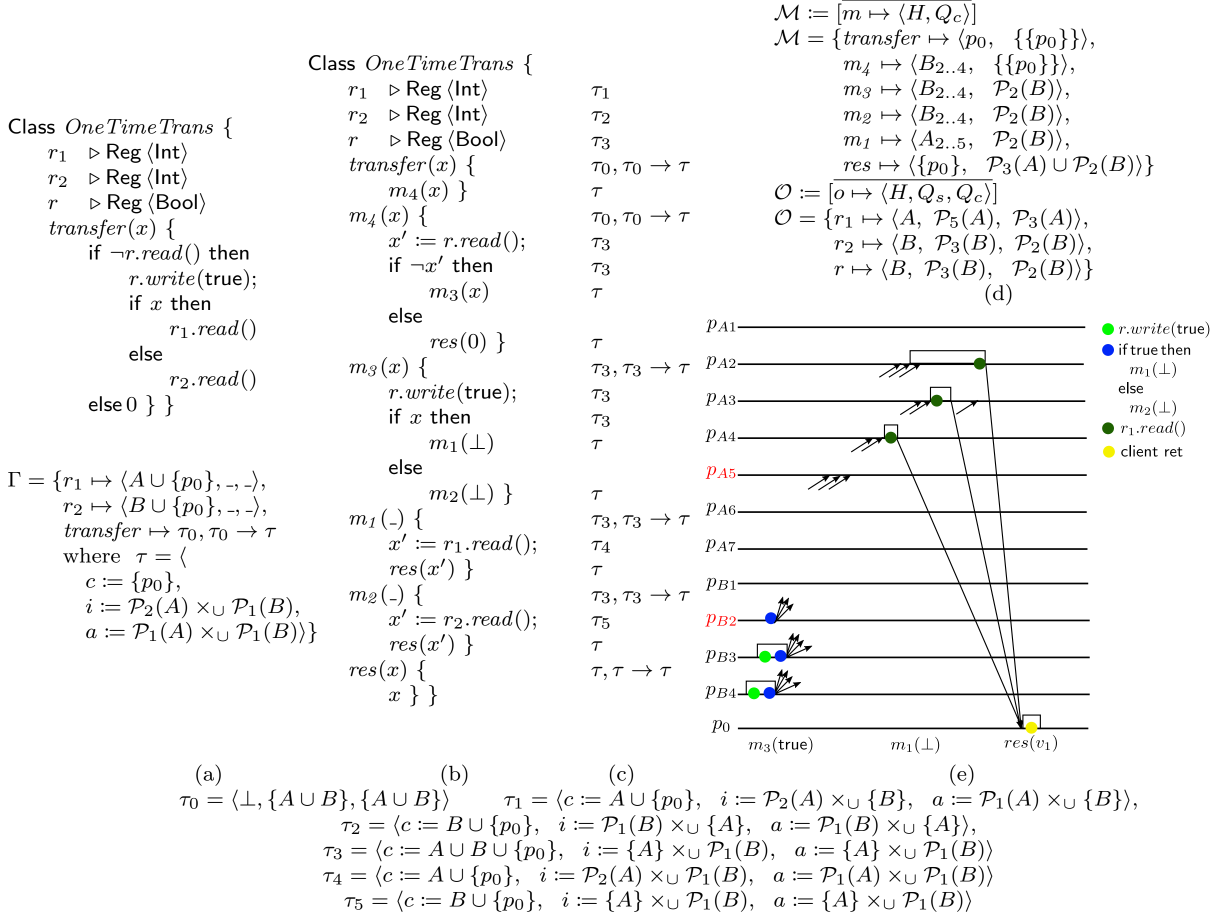


Figure 2.1: One-Time Transfer. (a) User specification, (b) Partitioned class, (c) Typing, (d) Placement, (e) Execution. $A = \{p_{A1}, .., p_{A7}\}$, $B = \{p_{B1}, .., p_{B4}\}$. The set $\mathcal{P}_n(S)$ is the set of subsets of S of cardinality n. The set $S_{i..j}$ denotes $\{p_{Si}, .., p_{Sj}\}$.

specifications, the type inference system can derive constraints for the placement and replication. Transforming and solving these constraints yields the Byzantine quorum systems that host the fields and and methods of the class. This leads to *trustworthy-by-construction distributed systems*: the user describes the class with type annotations specifying the trustworthiness policies, and our tool, HAMRAZ, automatically synthesizes a distributed system that assures the policies. HAMRAZ can automatically construct hosting Byzantine quorum systems, and adjust them to the resiliency strength of type specifications. Experimental results on a cluster of nodes show that HAMRAZ generates resilient systems; the resulting systems can gracefully tolerate attacks that are as strong as the specifications.

We will start with an overview in section 5.2. We see the programming model, the lattices of the security types, and quorum systems in section 2.3. Then, we present the partitioning transformations in section 2.4. Next, we see the operational semantics, the type inference system, and the security guarantee theorems in section 2.5, section 2.6, and section 2.7. Then, we consider constraint solving for type inference in section 2.8. We describe the implementation and report experimental results in section 2.9. Finally, we discuss the related works in section 2.10 before we conclude in section 2.11.

2.2 Overview

In this section, we see a glimpse of security types, partitioning, and the inference of placement and replication.

One-time Transfer. We illustrate these concepts by a simple use-case: *one-time transfer*: Alice manages the set of servers (or principals) $A = \{p_{A1}, p_{A2}, .., p_{A7}\}$ and a

register object r_1. Similarly, Bob manages the set of principals $B = \{p_{B1}, p_{B2}, .., p_{B4}\}$ and a register object r_2. The problem is to program a *transfer* method that lets a client principal p_0 (not Alice or Bob principals) choose to see *one of the two* registers, and reveals the value of that register to p_0 *only once*. The system should keep Alice's register *confidential* from Bob and vice versa. Further, there are *resiliency specifications for the integrity and availability* of the system. The system should maintain the integrity of the *transfer* method to return the correct value even if two of Alice's principals and one of Bob's principals are Byzantine. It should also stay available even if one of Alice's principals and one of Bob's principals are Byzantine. Byzantine principals are compromised by the adversary and may behave arbitrarily. (A variant where Alice and Bob are not authorized to view the client's choice is called oblivious transfer and is a use-case in our experiments.)

Centralized Definition. The one-time transfer class *OneTimeTrans* is shown in Figure 2.1.(a). It is a *high-level centralized definition* with no extra details for distribution. The class has three field objects: Alice's register r_1, Bob's register r_2, and the register r that stores whether the client has already read a value. The *transfer* method takes the client's choice as a boolean parameter x. It first checks the value of the register r. If it is `true`, the client has already read a value, and *transfer* simply returns 0. Otherwise, it sets r to `true`, and depending on the parity of x, it reads and returns either the value of the register r_1 or r_2.

Security Types. The user can also annotate her program with security types. A type is the triple $\langle c, i, a \rangle$ of confidentiality c, integrity i and availability a types. (In order to focus the type system on trustworthiness, types do not capture the classical representation

types such as Int.) The confidentiality type c of a value is the set of principals that are trusted to access the value. An integrity (and similarly availability) type represents resiliency against certain attacks. A *resiliency* value $\{\bar{b}\}$ characterizes all the attack scenarios of the system: for every execution of the system, there is a set b that contains all the Byzantine principals in that execution. The integrity type i of a value is defined as a resiliency $\{\bar{b}\}$ such that the value is correct even in the face of each Byzantine set b. Similarly, the availability type a of a value is a resiliency $\{\bar{b}\}$ such that the value is accessible even in the face of each b. For two resiliency values B and B', let their join $B \times_{\cup} B'$ be $\{b \cup b' \mid \langle b, b' \rangle \in B \times B'\}$. Types form a lattice with the weakest and strongest types $\perp$ and $\top$. In particular, a resiliency value B is stronger than (or can flow to) another B', written as $B \sqsubseteq B'$, if for any attack in B', there is a stronger attack in B. Let $\mathcal{P}_n(S)$ represent the set of subsets of the set S of cardinality n. For example, $\mathcal{P}_2(A)$ represents the attack scenario where two principals of the set A can be Byzantine.

Trustworthiness Policies. The user specifies the trustworthiness policies as type annotations. The type environment Γ in Figure 2.1.(a) represents the *user type annotations* for the *OneTimeTrans* class. The register r_1 should be confidential and accessible to only the set of principals A and the client p_0. Similarly, r_2 should be confidential for B and p_0. (The unspecified integrity and availability types are left for type inference as $_$.) The type of the *transfer* method is a function type $\tau_{\mathsf{x}}, \tau \rightarrow \tau'$ from the context type τ_{x} and the parameter type τ to the return type τ'. (As we will see later in section 2.6, the context type represents the trustworthiness of the calling context. Here, it is simply the type τ_0 as any context. Similarly, the parameter x is not confidential, has complete integrity and

availability at the client p_0; thus, its type is simply τ_0.) The return type τ of *transfer* is more interesting. The confidentiality type c is $\{p_0\}$; only the client p_0 should be able to call the method. The integrity type i is $\mathcal{P}_2(A) \times_\cup \mathcal{P}_1(B)$; it requires the integrity of the return value to be resilient to two A and one B Byzantine principals. Similarly, the availability type a is $\mathcal{P}_1(A) \times_\cup \mathcal{P}_1(B)$; it requires the availability to be resilient to one A and one B Byzantine principals. The goal is to automatically partition and replicate the field objects and methods so that the above specifications are satisfied.

Partitioning. The method *transfer* calls methods on both registers r_1 and r_2. However, there is no principal in the sets A and B that is authorized to see the values of both registers. Therefore, in section 2.4, we *adapt the CPS transformation* to partition the methods of a class to smaller methods such that each makes at most one call to an object. The result of partitioning the *transfer* method of Figure 2.1.(a) is presented in Figure 2.1.(b). The *transfer* method is partitioned into six methods. The earlier methods call later ones as tail-calls. The initial method *transfer* and the response method *res* are both hosted at the client p_0 to invoke the call and to later receive the return value respectively. (For uniformity, methods with no parameters take a dummy parameter _.) It is critical that the two calls on the two registers r_1 and r_2 are partitioned into the two separate methods m_1 and m_2; the type inference places them on separate sets of A and B hosts.

Replication. To satisfy the resiliency specifications, the field objects and the methods of the class should be sufficiently replicated. We apply Byzantine quorum systems for replication. A *quorum system* Q is a set of quorums; a quorum q is a set of principals that is adequate to properly perform operations. We use two types of quorum

systems: *communication quorum systems* and *storage quorum systems.* We use the former to communicate and validate method call requests, and the latter to replicate field objects. The placement $\mathcal{M}(m)$ of each method m is a pair $\langle H, Q_c \rangle$: the method m is replicated on the set of hosts H that each execute a call on m if they receive the same call from the communication quorum system Q_c. The placement $\mathcal{O}(o)$ of each field object o is a triple $\langle H, Q_s, Q_c \rangle$: the object o is replicated on the set of hosts H with the the storage quorum system Q_s that executes a method call on o if it receives the same call from the communication quorum system Q_c.

Consider a quorum system $Q = \{\overline{q}\}$ and a set of Byzantine principals b. As we will see in section 2.3, for the *availability of* Q, at least one q should not intersect with b. For the *integrity of* Q *as a storage system*, the intersection of every pair of quorums q_1 and q_2 should not be contained in b, and for the *integrity of* Q *as a communication system*, no q should be contained in b. The placements $\mathcal{O}$ and $\mathcal{M}$ for the *OneTimeTrans* class are shown in Figure 2.1.(d). (We will see below how these placements can be inferred from the specified policies.) The notation $S_{i..j}$ denotes the set $\{p_{Si}, .., p_{Sj}\}$. As an example, assume that we expect resiliency to two Byzantine principals in A. The storage quorum system Q_s for the register r_1 is $\mathcal{P}_5(A)$, subsets of A of size 5. The set A has 7 principals. Thus, there is always a quorum (a subset of A with size $7 - 2 = 5$) of non-Byzantine principals. Therefore, Q_s preserves its availability. Further, any pair of quorums intersect in at least $2 \times 5 - 7 = 3$ principals. Therefore, there is at least $3 - 2 = 1$ non-Byzantine principal in the intersection, and Q_s preserves its integrity. The communication quorum system Q_c for r_1 is $\mathcal{P}_3(A)$, subsets of A of size 3. Therefore, there is at least $3 - 2 = 1$ non-Byzantine

principal in every quorum. Thus, every call request received from a quorum is valid, and Q_c preserves its integrity.

We now consider a distributed execution with the given placements, and then consider placement inference.

Replication Semantics. An example execution fragment from the method call $m_3(\mathsf{true})$ to m_1 and finally *res* is shown in Figure 2.1.(e). The two principals p_{A5} and p_{B2} are Byzantine. The method m_3 is hosted on $B_{2..4}$. The non-Byzantine hosting principals of m_3 (i.e., p_{B3} and p_{B4}) execute $r.write(\mathsf{true})$ and the then branch of the subsequent if expression to call m_1. They send request messages to call m_1 to the hosting principals of m_1 that are $A_{2..5}$. A call to m_1 is executed only when the request is received from a quorum in $\mathcal{P}_2(B)$ (that is two B principals). Since there are enough non-Byzantine hosts for m_3, enough requests are received at hosts of m_1, and its non-Byzantine hosts (i.e., p_{A2}, p_{A3} and p_{A4}) execute m_1. The method m_1 reads the register r_1 and calls the method *res* with the read value. A call to *res* is executed only when the request is received from a quorum in $\mathcal{P}_3(A) \cup \mathcal{P}_2(B)$. In this case, three A principals make a quorum, and *res* is finally executed at the client p_0 with the value of the first register. We note that the quorums $\mathcal{P}_2(B)$ in the communication quorum system of *res* are used when the method m_2 calls *res*.

Type and Placement Inference. The user-specified type for the return value of *transfer* is τ. Therefore, the return type and the parameter of the response method *res* are expected to be of type τ. Given the type specification of the method *res*, and the user type annotations in Γ, the type inference system can *infer the types* of the other methods. Further, it can *infer the placement* for the field objects and the methods. We will see the

type inference system in section 2.6. The inferred type of each method and expression in Figure 2.1.(b) is written in front of it in Figure 2.1.(c). The inferred placements $\mathcal{O}$ and $\mathcal{M}$ are shown in Figure 2.1.(d). We look at a few steps that infer the storage quorum system of the object r_1, the communication quorum system of the method *res*, and the hosts of the method m_1.

Storage Quorum Inference. The type of the parameter x of the method *res* is τ, The integrity and availability components of τ are $i = \mathcal{P}_2(A) \times_{\cup} \mathcal{P}_1(B)$, and $a = \mathcal{P}_1(A) \times_{\cup} \mathcal{P}_1(B)$. The method m_1 calls *res* with the argument x'. The integrity $i_{x'}$ of the argument x' should be stronger than the integrity i of the parameter x, i.e., $i_{x'} \sqsubseteq i$. The variable x' in m_1 binds the return value of a call to the object r_1. Therefore, the integrity i_{r_1} of r_1 should be stronger than the integrity $i_{x'}$ of x', i.e., $i_{r_1} \sqsubseteq i_{x'}$. The integrity of r_1 is determined by the quorum system Q_s that stores it. By the transitivity of the above relations, the integrity i_{Q_s} of Q_s should be stronger than the integrity i above, i.e., $i_{Q_s} \sqsubseteq i$. A similar argument for the availability a_{Q_s} of Q_s yields $a_{Q_s} \sqsubseteq a$. Further, according to the confidentiality of r_1, Q_s should be stored on only the A principals. Therefore, the integrity of Q_s should be resilient to 2 Byzantine principals, and the availability of Q_s should be resilient to 1 Byzantine principal. What is the size s of the subset of A that hosts Q_s? Further, what is the quorum size n for Q_s? The quorum system Q_s will be $\mathcal{P}_n(A_{1..s})$. For integrity, the quorums should have a non-Byzantine intersection. Thus, we should have $2 \times n - s > 2$. For availability, there should be a non-Byzantine quorum. Thus, we should have $s - 1 \geq n$. A solution to these constraints is $s = 7$ and $n = 5$. As Figure 2.1.(d) shows, the storage quorum system Q_s for r_1 is $\mathcal{P}_5(A)$ (as $A_{1..7}$ is obviously equal to A).

Communication Quorum Inference. The integrity type of the parameter x of *res* is $i = \mathcal{P}_2(A) \times_\cup \mathcal{P}_1(B)$. The quorum system Q_c that receives calls to *res* should have stronger integrity than the parameter. Thus, its integrity should be resilient to 2 A and 1 B Byzantine principals. Therefore, the quorums in Q_c should have at least $2+1 = 3$ principals from A, or $1+1 = 2$ principals from B. Therefore, as the placement $\mathcal{M}$ in Figure 2.1.(d) shows, Q_c is $\mathcal{P}_3(A) \cup \mathcal{P}_2(B)$.

Host Inference. Now, let us consider the hosting principals of m_1. Since m_1 calls a method on r_1, and r_1 is confidential for A, the method m_1 can be hosted on only A principals. The method m_1 calls *res*. As we just saw, the quorum system Q_c of *res* receives a call from A subsets of size 3. The type of the parameter x of *res* is τ, and the availability component of τ is $a = \mathcal{P}_1(A) \times_\cup \mathcal{P}_1(B)$. The parameter should be available if one principal in A is Byzantine. To satisfy the availability of the parameter of *res*, the method m_1 should send the argument from at least $3+1 = 4$ principals in A. Therefore, the hosting principals of m_1 are a subset of A of size 4. The placement shown in Figure 2.1.(d) chose the set $A_{2..5}$.

2.3 Classes and Security Types

Class Definition. As shown in Figure 2.2, a class $C = \langle \overline{o}, \overline{d} \rangle$ is described as a set of field objects o and method definitions d. A method definition $m(x) \coloneqq e$ defines a method m with parameter x and the body e. An expression e is either an integer value v or none value $\bot$, a variable x, an operation $\oplus$ on expressions, a sequence expression $x \coloneqq e; e'$ that evaluates e and binds its value to x for e', a conditional expression if, a this-method call $m(e)$ (a method call on the current object this of the enclosing class), or an object-method

C	$:=$	$\langle \overline{o}, \overline{d} \rangle$	Class
o			Field Object
d	$:=$	$m(x) := e$	Method Definition
e	$:=$	$v \mid \bot \mid x \mid e \oplus e \mid x := e; e$	Expression
	$\mid$	$\text{if } e \text{ then } e \text{ else } e \mid m(e) \mid o.m(e)$	
p, h	$:$	P	Principal or Host
q, b, H	$:$	$\mathcal{P}(P)$	Quorum, Byzantine set, Hosts
Q, B	$:$	$\mathcal{P}(\mathcal{P}(P))$	Quorum System, Resiliency
$\mathcal{M}$	$:=$	$\overline{[m \mapsto \langle H, Q_c \rangle]}$	Method Placement
$\mathcal{O}$	$:=$	$\overline{[o \mapsto \langle H, Q_s, Q_c \rangle]}$	Object Placement
τ	$:=$	$\langle c, i, a \rangle$	Type
c	$:=$	H	Confidentiality
i	$:=$	B	Integrity
a	$:=$	B	Availability

Figure 2.2: Syntax. $\mathcal{P}(S) = 2^S$ is the power set of S.

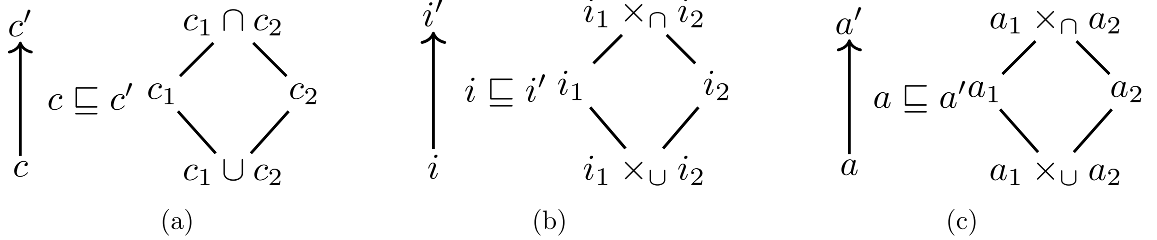


Figure 2.3: Lattices for (a) Confidentiality (b) Integrity and (c) Availability. Arrows show the correct flow direction: more confidentiality, less integrity, and less availability.

call $o.m(e)$. The sequence expression $e; e'$ is a syntactic sugar for a sequence whose bound variable is not free in e'. The language achieves Turing-completeness through recursive this-method calls.

Principal Sets. A principal (process or host) denoted as p (or h) from the universe of principals P can host both objects and methods. The identity of principals can be authenticated. A quorum q, Byzantine set b, or hosts H is a set of principals. A quorum system Q, or resiliency value B is a set of subsets of principals (such that none of the subsets is contained in another). A quorum system $\{\overline{q}\}$ is a set of quorums q such that each is adequate to perform operations. A resiliency value $\{\overline{b}\}$ characterizes all the

attack scenarios of the system: for every execution, at least one of the b sets contains all the Byzantine principals in that execution. Byzantine principals are controlled by the adversary; they may not follow the user-defined programs and system protocols. (In the literature, a set $\{\bar{b}\}$ is called a failure-prone system [326] as well.)

We define the basic operators $\sqsubseteq$, $\times_\cup$ and $\times_\cap$ on quorum systems and resiliency values. A resiliency value B is stronger than (or can flow to) another B', written as $B \sqsubseteq B'$ iff for every set b' in B', there is a set b in B such that $b' \subseteq b$. We say that a Byzantine attack b is subsumed by a resiliency value B iff $B \sqsubseteq \{b\}$. Finally, $B \times_\cup B' = \{b \cup b' \mid \langle b, b' \rangle \in B \times B'\}$ and similarly, $B \times_\cap B' = \{b \cap b' \mid \langle b, b' \rangle \in B \times B'\}$.

Method and Object Placement. A method placement $\mathcal{M}$ is a mapping from each method m to a pair $\langle H, Q_c \rangle$ where H is the set of principals that host m, and Q_c is the communication quorum system for requests to call m. A host h in H executes a call to m only if it receives the call with the same argument from a quorum q in Q_c. An object placement $\mathcal{O}$ is a mapping from each object o to a triple $\langle H, Q_s, Q_c \rangle$ where H is the set of principals that hosts o, Q_s is the storage quorum system that serves calls to o, and Q_c is the communication quorum system to request calls on o. A method call on o is executed by a quorum in Q_s only if they receive the same call from a quorum in Q_c. (We will see more details of the operational semantics in section 2.5.)

Security Types. A type τ is a tuple $\langle c, i, a \rangle$ where c is the confidentiality, i is the integrity, and a is the availability type.

Confidentiality. The confidentiality type c of a value is the set of principals that are trusted to access the value. A confidentiality type c is less than (or can flow to)

another c', written as $c \sqsubseteq c'$, iff $c' \subseteq c$. As Figure 2.3.(a) shows, information can flow from low confidentiality c to high confidentiality c'. Assume that the confidentiality type of x is $c_x = \{p_1, p_2\}$ and the confidentiality type of x' is $c_{x'} = \{p_1\}$. $c_x \sqsubseteq c_{x'}$. The flow from x to x' leaks no information, but the flow from x' to x can leak a secret in x' to p_2. Confidentiality types form a lattice where join $\sqcup$ is $\cap$, meet $\sqcap$ is $\cup$, $\bot$ is P and $\top$ in $\emptyset$.

Integrity. The integrity type i of a value is defined as a resiliency $\{\bar{b}\}$ such that the value is correct even in the face of each Byzantine attack b. We say that an integrity type i is stronger than (or can flow to) another i' if $i \sqsubseteq i'$. (We saw the definition of $\sqsubseteq$ on resiliency values above.) Intuitively, larger Byzantine sets b represent more integrity. As Figure 2.3.(b) shows, information can flow from high integrity i to low integrity i'. Assume that the integrity type of x is $i_x = \{\{p_1\}, \{p_2\}\}$ and the integrity type of x' is $i_{x'} = \{\{p_1, p_2, p_3\}\}$. The variable x preserves its integrity even if p_1 or p_2 are Byzantine. The variable x' preserves its integrity even if p_1, p_2 and p_3 are Byzantine. Thus, $i_{x'} \sqsubseteq i_x$. The flow from x' to x preserves the integrity of x. However, the flow from x to x' can violate the integrity of x' if both p_1 and p_2 are Byzantine, or p_3 is Byzantine. Integrity types form a lattice where join $\sqcup$ is $\times_\cap$, meet $\sqcap$ is $\times_\cup$, $\bot$ is $\{P\}$ and $\top$ in $\{\emptyset\}$.

Availability. Similar to integrity, the availability type a of a value is defined as a resiliency $\{\bar{b}\}$ such that the value is accessible even in the face of each Byzantine attack b. We say that an availability type a is stronger than (or can flow to) another a' if $a \sqsubseteq a'$. As Figure 2.3.(c) shows, information can flow from high availability a to low availability a'. Similar to integrity, if we have $a_{x'} \sqsubseteq a_x$, the flow from x' to x preserves the availability of x but not vice versa. Availability types form a lattice where join $\sqcup$ is $\times_\cap$, meet $\sqcap$ is $\times_\cup$,

$\perp$ is $\{P\}$ and $\top$ in $\{\emptyset\}$. We note that no resiliency is represented as $\{\emptyset\}$. An integrity or availability type is expected to be non-empty.

We represent and analyze integrity and availability types separately. However, an available value is often usable only if it has integrity. To assure availability of a correct value, the Byzantine set should be subsumed by both the integrity and availability types. Therefore, as Figure 2.1.(a) shows, integrity type is often stronger than availability type.

Type. A type $\tau = \langle c, i, a \rangle$ is a subtype of another type $\tau' = \langle c', i', a' \rangle$, written as $\tau \sqsubseteq \tau'$, iff $c \sqsubseteq c'$, $i \sqsubseteq i'$ and $a \sqsubseteq a'$. We note that with the lattice (and flow) direction defined for integrity and availability above, all the three type components are co-variant. Intuitively, the super-type has more confidentiality, less integrity and less availability. A type can be implicitly up-cast to a super-type. If $\tau_x \sqsubseteq \tau_{x'}$ then it is safe for the data from x to flow into x'. Types form a lattice with the expected point-wise definitions for $\sqcup$, $\sqcap$, $\perp$ and $\top$ on the lattices of their three components.

Resiliency of Quorum Systems. First, we consider the integrity of communication and storage in turn. Then, we consider the availability of quorum systems.

Integrity of Communication Quorum Systems. Communication quorum systems are used to deliver a message to a target principal. Sender principals echo the message, and the target principal delivers it only if it receives the same message from a quorum. Thanks to the redundancy in the messages, the delivered message has integrity even if only one of the senders is a non-Byzantine principal. The integrity of a communication quorum system $Q = \{\overline{q}\}$, written as *CIntegrity*(Q), is the resiliency $B = \{\bar{b}\}$: the Byzantine sets b are the maximal subsets of the set of principals P, for each b, no quorum q that is a subset of b.

Integrity of Storage Quorum Systems. Storage quorum systems are used to store and retrieve objects. To store a value for the object, at least a quorum should store it, and to retrieve its value, at least a quorum should retrieve the same value. In order to retrieve the latest stored value, it is crucial that the two quorums have a non-Byzantine principal in their intersection. The integrity of a storage quorum system $Q = \{\overline{q}\}$, written as $\mathit{SIntegrity}(Q)$, is the resiliency $B = \{\bar{b}\}$ where the Byzantine sets b are the maximal subsets of the set of principals P such that for each b, the intersection of every pair of quorums q_1 and q_2 is not a subset of b.

Availability of Quorum Systems. Given a set of hosts H for a quorum system $Q = \{\overline{q}\}$ and a set of Byzantine principals b, consider a quorum q that is tasked with the execution of an operation. The quorum q can perform the operation if all of its members are in H and none of them are in b. Therefore, Q is available if at least one of its quorums q is a subset of H and doesn't intersect b. The availability of a set of hosts H for a quorum system $Q = \{\overline{q}\}$, written as $\mathit{Availability}(Q, H)$, is the resiliency $B = \{\bar{b}\}$ where the Byzantine sets b are the maximal subsets of the set of principals P such that for each b, there is at least one quorum q that is a subset of H and doesn't intersect b.

We note that as a quorum system is a set of quorums, the classical labels that represent one set of principals are not enough to capture its integrity and availability.

2.4 Partitioning

A method of a class can execute multiple calls on its field objects. A principal that hosts a method should be more confidential than all the objects that it accesses. Such

$$
\begin{array}{rcll}
f & := & v \mid \bot & \text{Integer Literal} \\
& \mid & x & \text{Variable} \\
& \mid & f \oplus f & \text{Operation} \\
& \mid & \mathsf{if}\ f\ \mathsf{then}\ f\ \mathsf{else}\ f & \text{Conditional} \\
& \mid & m(f) & \text{This call} \\
c & := & \mathsf{call}\ x := o.m(f)\ \mathbf{in}\ c & \text{Method Call} \\
& \mid & \mathsf{if}\ f\ \mathsf{then}\ c\ \mathsf{else}\ c & \text{Conditional} \\
& \mid & f & \text{Expression}
\end{array}
$$

(a)

```
call x1 := r.read() in
if ¬x1 then
  call _ := r.write(true) in
  if x then
    call x2 := r1.read() in
    res(x2)
  else
    call x3 := r2.read() in
    res(x3)
else res(0)
```

(b)

$$
\begin{array}{rcl}
[\![v]\!]\,k & = & k\,v \\
[\![\bot]\!]\,k & = & k\,\bot \\
[\![x]\!]\,k & = & k\,x \\
[\![e_1 \oplus e_2]\!]\,k & = & [\![e_1]\!]\,(\lambda x_1.\,[\![e_2]\!]\,(\lambda x_2.\,k\,(x_1 \oplus x_2))) \\
[\![\mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2]\!]\,k & = & [\![e_0]\!]\,(\lambda x.\,\mathsf{if}\ x\ \mathsf{then}\ [\![e_1]\!]\,k\ \mathsf{else}\ [\![e_2]\!]\,k) \\
[\![x := e_1; e_2]\!]\,k & = & [\![e_1]\!]\,(\lambda x.\,[\![e_2]\!](\lambda x_2.\,k\,x_2)) \\
[\![m(e)]\!]\,k & = & [\![e]\!]\,(\lambda x.\ k\ m(x)) \\
[\![o.m(e)]\!]\,k & = & [\![e]\!]\,(\lambda x_1.\,\mathsf{call}\ x_2 := o.m(x_1)\ \mathbf{in}\ k\,x_2)
\end{array}
$$

(c)

Figure 2.4: Factoring. (a) Factored expressions. (b) Example factored expression $[\![\mathit{transfer}\ (x)]\!]\ \mathit{res}$. (c) CPS transformation.

a principal might not exist. For example, in our running example, the one-time transfer class *OneTimeTrans* in Figure 2.1.(a), the method *transfer* calls methods on both objects r_1 and r_2. However, there is no principal (except the client) that is confidential enough to access both objects. Further, placing more methods on more confidential principals can cause imbalance for the computation load across principals. Therefore, as the first step for resilient replication, we partition the methods of the class into smaller methods such that each method makes at most one object-method call. When methods make at most one object-method call, they provide maximum flexibility for their placement and replication during the type inference.

We perform partitioning in two steps: we first factor the object-method calls by an adaptation of the CPS transformation, and then split the methods.

Factoring Object-method Calls. We first transform expressions e (that we saw in Figure 2.2) to factored expressions c as defined in Figure 2.4.(a). In a factored expression c, object-method calls are lifted as explicit call expressions $\mathsf{call}\, x := o.m(f)$ where the expressions f are free of object-method calls. As an example, Figure 2.4.(b) shows the factored representation of the body of the *transfer* method that we saw in Figure 2.1.(a). The object-method call $r.\mathit{read}()$ is lifted to the beginning of the body. Similarly, $r_1.\mathit{read}()$

and $r_2.read()$ are lifted to the beginning of their enclosing branches of the if statement. Factoring is performed in two steps: First, we perform a CPS transformation to make object-method calls and their evaluation order explicit. Object-method calls are translated to explicit call expressions, and their call-by-value order of evaluation is made explicit as nested call expressions. Second, once the call expressions are captured, we apply β-reduction to remove redundant lambda abstractions.

CPS Transformation. The CPS transformation is presented in Figure 2.4.(c). The rules for values v and $\bot$, variables x, and operations $\oplus$ are straightforward. The rule for the if expression applies the transformation to the condition and the branches in order. The rule for the sequence expressions $x \coloneqq e_1; e_2$ first evaluates the first expression e_1, and then passes its result as x to the second expression e_2 and evaluates it. The rule for this-method calls $m(e)$ evaluates the argument e before calling the method m. Similarly, the rule for object-method calls $o.m(e)$ evaluates the argument e first. More importantly, it converts $o.m(e)$ to an explicit call expression call $x_2 \coloneqq o.m(x_1)$ **in** $k\, x_2$ instead of directly passing it to the continuation k as $k\, o.m(x_1)$. The explicit call expressions preserve object-method calls and their order after β-reductions.

Reduction. The CPS transformation introduces *administrative* lambda abstractions to pass continuations and make the evaluation order explicit. After the transformation, we apply β-reduction to remove the administrative lambda abstractions and restore a concise representation.

$$\beta\text{-REDUCTION} \quad (\lambda x.\, e)\, e' \rightarrow e[e'/x]$$

We note that the β-reduction cannot rewrite call expressions. Consider the expression

$$\textsc{Call}\quad \frac{[\![c]\!] \triangleright \langle f', X', D' \rangle \qquad \text{fresh}\, m' \qquad X'' = \mathrm{FV}(f) \cup X' \setminus x}{[\![\mathsf{call}\ x \coloneqq o.m(f)\ \mathbf{in}\ c]\!] \triangleright \langle m'(X''),\ X'',\ D' \cup \{m'(X'') \coloneqq (x \coloneqq o.m(f); f')\} \rangle}$$

$$\textsc{If}\quad \frac{[\![c_1]\!] \triangleright \langle f_1, X_1, D_1 \rangle \quad [\![c_2]\!] \triangleright \langle f_2, X_2, D_2 \rangle}{[\![\mathsf{if}\ f\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!] \triangleright \langle \mathsf{if}\ f\ \mathsf{then}\ f_1\ \mathsf{else}\ f_2, \mathrm{FV}(f) \cup X_1 \cup X_2, D_1 \cup D_2 \rangle}$$

$$\textsc{FExp}\quad [\![f]\!] \triangleright \langle f, \mathrm{FV}(f), \emptyset \rangle$$

$$\textsc{Method}\quad \frac{[\![e]\!] \triangleright \langle e', _, D \rangle}{[\![m(x) \coloneqq e]\!] \triangleright D \cup \{m(x) \coloneqq e'\}}$$

$$\textsc{Class}\quad [\![\langle \overline{o}, \overline{d} \rangle]\!] \triangleright \langle \overline{o}, \cup \overline{[\![d]\!]} \rangle$$

Figure 2.5: Splitting Transformation

call $x_2 \coloneqq o.m(x_1)$ **in** $k\, x_2$. The β-reduction cannot replace x_2 with $o.m(x_1)$. Therefore, even if x_2 is not used in k, the object-method call $o.m(x_1)$ is not removed. Further, the evaluation order of calls is preserved.

Splitting This-methods. Given a class such that the body of the this-methods are factored expressions c, this step splits each this-method into multiple this-methods such that each one calls at most one object-method call.

Figure 2.5 presents the splitting transformation. The judgments that translate a factored expression c are of the form $[\![c]\!] \triangleright \langle f, X, D \rangle$ where f is the resulting expression, X is the set of free variables of f, and D is the set of generated this-method definitions that f transitively calls. The rule Call translates a call expression call $x \coloneqq o.m(f)$ **in** c. It first translates c to f' with free variables X' and this-methods D'. It then generates a new this-method m' with the sequence $x \coloneqq o.m(f); f'$ as the body. We note that the generated this-method includes only one object-method call. The free variable X'' in the body are the free variables of f and the free variables of f' (i.e., X') except x that is bound by the call. The free variables X'' should be passed as parameters to m'. Thus, the call

expression is translated to the this-method call $m'(X'')$. If the translated call expression is part of a larger expression, the resulting this-method call $m'(X'')$ can inductively become a leaf expression of the body of the this-method that is generated for the larger expression. Thus, calls to the generated this-methods appear only as tail-calls. The rule IF inductively splits the branches of the if expression and results in an if statement that is free of object-method calls. The rule FEXP simply translates an f expression to the tuple of itself, its free variables FV(f) and no new this-methods. The rule METHOD splits a this-method, and the rule CLASS splits each this-method of a class.

As an example, applying the splitting transformation to the factored expression in Figure 2.4.(b) results in the split methods in Figure 2.1.(b). Starting from the leaf expressions, the two branches of the inner if expression are split to the two methods m_1 and m_2. This split is crucial to the enforcement of the confidentiality policies. The objects r_1 and r_2 can be accessed by only the principals A and B respectively. After the split, the methods m_1 and m_2 can be separately placed on A and B principals. Next, the then branch of the outer if expression is split to the method m_3. We note that there is only one object-method call at the beginning of each generated method. Next, the outer if expression is split to the method m_4 where the object-method call $r.read()$ is at the top. Finally, the body of the top-level this-method *transfer* is translated to a call to m_4.

We note that partitioning allows maximum flexibility for placement inference; however, if a caller and a callee are placed on the same hosts, the call can be inlined. We also note that partitioning results in non-blocking methods where a call to the method immediately returns, and the return value is later passed by a callback method (e.g. *res*).

If needed, a non-blocking method can be made blocking by simply waiting for the callback using standard synchronization mechanisms.

2.5 Operational Semantics

In the previous section, we saw how the methods of a class are partitioned. In this section, we present the operational semantics of these classes. We first present a baseline sequential semantics and then present a distributed semantics that replicates both objects and methods, and marshals call requests between quorums of hosts.

Sequential Semantics. The state of the operational semantics is defined in Figure 2.6. For the sequential semantics, the state is the pair $\langle e, S\rangle$ where e is an expression, and S is a mapping from objects o to their encapsulated states s. A reduction context $\mathcal{R}$ captures the next expression to be reduced. The sequential operational semantics is presented in Figure 2.7. The transitions are straightforward. We saw in the previous section that after partitioning, this-method calls appear only as tail-calls. Thus, in the rule SThisCall, this-method calls do not need a context $\mathcal{R}$. For brevity, we use $_$ in the place of symbols that are unused and stay the same in the transition. (In order to factor the reduction context, an extra rule could be added. For the next semantics, yet another rule would be needed to factor the context for multiple principals. The current representation with explicit contexts seems to be more concise.)

Distributed Semantics. The distributed state is the triple $\langle \mathbb{P}, \mathbb{S}, \mathbb{N}\rangle$. As defined in Figure 2.6, the principal states $\mathbb{P}$ is a map from principals to their states. The state of a principal is the expression e that it is executing, the identifier id of the call that is being

executed, and the number of calls n that have been executed by this call. Consider the call tree rooted at the initial method call. The unique identifier of a call is the list of the branch numbers in the path from the initial method call to that call. The object state $\mathbb{S}$ is a map from objects o to pairs $\langle s, r \rangle$ where s is the state of o, and r is the recorded calls, a map from call identifiers to their return values. Consider a this-method $m()$ that calls an object-method $o.m'()$. Since $m()$ is replicated on multiple principals, multiple requests to execute $o.m'()$ with the same identifier can be issued. To avoid duplicate execution of object-method calls, the storage quorum system of an object o not only keeps its state s but also a record r of the previously executed calls and their return values. The network $\mathbb{N}$ keeps pending request messages to execute this-method calls. It is a mapping from tuples $\langle p, id, m, v \rangle$ to a set of principals q (or $\perp$), where p is the receiver principal, id is the identifier of the call, m is the method requested to be called, v is the argument, and q is the set of sender principals that requested the call. The value of the mapping is $\perp$ when the requested call is already processed and should not be reprocessed. A transition label and a sequence of labels are denoted as l and L respectively.

Figure 2.8 defines the distributed operational semantics that models the runtime system. It is parametric in terms of a class $C = \langle \overline{o}, \overline{d} \rangle$, the method and object placements $\mathcal{M}$ and $\mathcal{O}$, and the set of Byzantine hosts $\mathcal{B}$. We say that the resiliency of a system is $\{\overline{b}\}$ if it is resilient in all executions of all instantiations of the operational semantics with a $\mathcal{B}$ where $\{\overline{b}\} \sqsubseteq \{\mathcal{B}\}$, that is a $\mathcal{B}$ that is a subset of a b. Thus, in the theorems of section 2.7, $\mathcal{B}$ is universally quantified.

$$
\begin{array}{rcll}
\langle e, S\rangle & & & \text{Sequential State} \\
\mathcal{R} & := & \mathcal{R} \oplus e \mid v \oplus \mathcal{R} \mid x := \mathcal{R};\ e & \text{Red. Context} \\
 & \mid & \textsf{if } \mathcal{R} \textsf{ then } e \textsf{ else } e \mid m(\mathcal{R}) \mid o.m(\mathcal{R}) \mid [\,] & \\
s & & & \text{Object State} \\
S & := & [\overline{o \mapsto s}] & \text{Seq. Object States} \\
\langle \mathbb{P}, \mathbb{S}, \mathbb{N}\rangle & & & \text{Distributed State} \\
\mathbb{P} & := & [\overline{p \mapsto \langle e, id, n\rangle}] & \text{Principal States} \\
id & : & \textsf{list nat} & \text{Method call identifier} \\
\mathbb{S} & := & [\overline{o \mapsto \langle s, r\rangle}] & \text{Dist. Object States} \\
r & := & [\overline{id \mapsto v}] & \text{Method calls record} \\
\mathbb{N} & := & [\overline{\langle p, id, m, v\rangle \mapsto q \mid \bot}] & \text{Network} \\
l & := & p \mid \overline{p} \mid \langle p, id, m, v\rangle & \text{Label} \\
 & \mid & \langle p, p, m, id, v\rangle \mid \langle p, v\rangle & \\
L & := & l^* & \text{Labels}
\end{array}
$$

Figure 2.6: Runtime State

$$
\textsc{SOp}\ \frac{v_1 \oplus v_2 = v_3}{\langle \mathcal{R}[v_1 \oplus v_2], _\rangle \to \langle \mathcal{R}[v_3], _\rangle}
\qquad
\textsc{SSeq}\ \langle \mathcal{R}[x := v; e], _\rangle \to \langle \mathcal{R}[e[v/x]], _\rangle
$$

$$
\textsc{SIfThen}\ \langle \mathcal{R}[\textsf{if } 1 \textsf{ then } e_1 \textsf{ else } e_2], _\rangle \to \langle \mathcal{R}[e_1], _\rangle
\qquad
\textsc{SIfElse}\ \langle \mathcal{R}[\textsf{if } 0 \textsf{ then } e_1 \textsf{ else } e_2], _\rangle \to \langle \mathcal{R}[e_2], _\rangle
$$

$$
\textsc{SThisCall}\ \frac{(m(x) := e) \in \overline{d}}{\langle m(v), _\rangle \to \langle e[v/x], _\rangle}
\qquad
\textsc{SObjCall}\ \frac{S(o) = s \qquad m(s, v) = \langle s', v'\rangle}{\langle \mathcal{R}[o.m(v)], S\rangle \to \langle \mathcal{R}[v'], S[o \mapsto s']\rangle}
$$

Figure 2.7: Sequential Operational Semantics

The rules Op, Seq, IfThen and IfElse make local transitions and are similar to their sequential counterparts.

This-method calls. The rule ThisCall reduces a this-method call $m(v)$ on a principal p. Let the identifier of the method call that is currently being executed be id, and the number of calls that it has executed be n. Thus, the identifier of the new method call is $id :: n$. Let the placement of m be the set of principals $\{\overline{p'}\}$. A request message from the sender p to execute the method m with identifier $id :: n$ and argument v is sent to every

Op

$$\frac{v_1 \oplus v_2 = v_3}{\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[v_1 \oplus v_2], _, _\rangle], _, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[v_3], _, _\rangle], _, _\rangle}$$

Seq

$$\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[x \coloneqq v; e], _, _\rangle], _, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[e[v/x]], _, _\rangle], _, _\rangle$$

IfThen

$$\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[\mathsf{if}\ 1\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2], _, _\rangle], _, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[e_1], _, _\rangle], _, _\rangle$$

IfElse

$$\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[\mathsf{if}\ 0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2], _, _\rangle], _, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[e_2], _, _\rangle], _, _\rangle$$

ThisCall

$$\frac{\mathcal{M}(m) = \langle \{\overline{p'}\}, _\rangle \quad \overline{\mathbb{N}(p', id :: n, m, v) = q} \quad \mathbb{N}' = \mathbb{N}[\overline{(p', id :: n, m, v) \mapsto q \cup \{p\}}]}{\langle \mathbb{P}[p \mapsto \langle m(v), id, n\rangle], _, \mathbb{N}\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \bot, id, n+1\rangle], _, \mathbb{N}'\rangle}$$

ThisCallExec

$$\frac{\mathcal{M}(m) = \langle _, \{\overline{q}\}\rangle \quad \mathbb{N}(p, id, m, v) = q' \quad q \subseteq q' \qquad (m(x) \coloneqq e) \in \overline{d} \quad \mathbb{N}' = \mathbb{N}[(p, id, m, v) \mapsto \bot]}{\langle \mathbb{P}[p \mapsto \langle \bot, _, _\rangle], _, \mathbb{N}\rangle \overset{\langle p, id, m, v\rangle}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle e[v/x], id, 0\rangle], _, \mathbb{N}'\rangle}$$

ThisCallByz

$$\frac{p \in \mathcal{B} \quad \mathbb{N}(p', id', m, v) = q' \quad \mathbb{N}' = \mathbb{N}[(p', id', m, v) \mapsto q' \cup \{p\}]}{\langle \mathbb{P}[p \mapsto \langle e, id, n\rangle], _, \mathbb{N}\rangle \overset{\langle p, p', id', m, v\rangle}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle e', id', n'\rangle], _, \mathbb{N}'\rangle}$$

ObjCall

$$\frac{\begin{array}{c}\mathcal{O}(o) = \langle H, Q_s, Q_c\rangle \quad Q_c = \{\overline{q}\} \quad q \subseteq \{\overline{p}\} \quad \mathit{SIntegrity}(Q_s) \sqsubseteq \{\mathcal{B}\} \quad \mathit{Availability}(Q_s, H) \sqsubseteq \{\mathcal{B}\} \\ \mathbb{S}(o) = \langle s, r\rangle \quad id :: n \notin \mathsf{dom}(r) \quad m(s, v) = \langle s', v'\rangle \quad \mathbb{S}' = \mathbb{S}[o \mapsto \langle s', r[id :: n \mapsto v']\rangle]\end{array}}{\langle \mathbb{P}[\overline{p \mapsto \langle \mathcal{R}[o.m(v)], id, n\rangle}], \mathbb{S}, _\rangle \overset{\overline{p}}{\rightsquigarrow} \langle \mathbb{P}[\overline{p \mapsto \langle \mathcal{R}[v'], id, n+1\rangle}], \mathbb{S}', _\rangle}$$

ObjReCall

$$\frac{\begin{array}{c}\mathcal{O}(o) = \langle H, Q_s, _\rangle \quad \mathit{SIntegrity}(Q_s) \sqsubseteq \{\mathcal{B}\} \\ \mathit{Availability}(Q_s, H) \sqsubseteq \{\mathcal{B}\} \quad \mathbb{S}(o) = \langle s, r\rangle \quad r(id :: n) = v'\end{array}}{\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[o.m(v)], id, n\rangle], \mathbb{S}, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[v'], id, n+1\rangle], \mathbb{S}, _\rangle}$$

ObjCallByz

$$\frac{\mathcal{O}(o) = \langle _, Q_s, _\rangle \quad \mathit{SIntegrity}(Q_s) \not\sqsubseteq \{\mathcal{B}\}}{\langle \mathbb{P}[p \mapsto \langle \mathcal{R}[o.m(v)], _, _\rangle], _, _\rangle \overset{p}{\rightsquigarrow} \langle \mathbb{P}[p \mapsto \langle \mathcal{R}[v'], _, _\rangle], _, _\rangle}$$

Figure 2.8: Distributed Operational Semantics. It is parametric in terms of the class $C = \langle \overline{o}, \overline{d}\rangle$, the method and object placements $\mathcal{M}$ and $\mathcal{O}$, and the Byzantine principals $\mathcal{B}$. In the ThisCall rule, the union operator $\cup$ is extended for $\bot$ values: $\bot \cup s = \bot$.

receiver p'. For each receiver p', the principal p is added to the set of senders q. As we saw in section 2.4, after partitioning, this-method calls appear only as tail-calls. Thus, they transfer control to the hosts of the callee, and the new expression of the hosting principal p of the caller becomes the none value $\perp$ which denotes that p can execute another this-method call. We saw an example of this-method calls in Figure 2.1.(e). The non-Byzantine hosting principals of m_3 (i.e., p_{B3} and p_{B4}) issue a call to m_1. They send call request messages to the hosting principals of m_1 (i.e., $A_{2..5}$).

The rule ThisCallExec executes a this-method call if request messages from a quorum are received. The receiving principal p is not processing any calls as its current expression is none $\perp$. Let the communication quorum system for the method m be $\{\overline{q}\}$. If the set of requesting principals q' to call $m(v)$ (with the same identifier) is a superset of a quorum q, the current expression of p becomes the body e of m after the parameter x is substituted with the argument v. Finally, the processed request message is mapped to $\perp$ to prevent duplicate executions. We saw an example in Figure 2.1.(e). The communication quorum system for m_1 is $\mathcal{P}_2(B)$; a quorum should have at least two B principals. The non-Byzantine hosts of m_1 (i.e., p_{A2}, p_{A3} and p_{A4}) executed m_1, since each received the request from a quorum.

The rule ThisCallByz models the behavior of Byzantine principals that can arbitrarily change their state, and send arbitrary call requests to arbitrary principals. However, thanks to authentication upon receiving messages, they cannot send a message on behalf of another principal. (A step by a Byzantine principal to impersonate another Byzantine principal can be simulated as two consecutive steps by the two principals.)

Object-method calls. The rule ObjCall executes an object-method call $o.m(v)$. Let the storage quorum system of o be Q_s, and the communication quorum system of o be $Q_c = \{\overline{q}\}$. If (1) a set of principals $\{\overline{p}\}$ that are a superset of a communication quorum q call the object-method, (2) the set of Byzantine principals $\mathcal{B}$ cannot compromise the integrity and availability of the storage system Q_s, and (3) the method call is not already executed, i.e., it is not in the recorded calls r, then the method call is executed on the current state s of o. The resulting state s' is stored, and the return value v' is recorded in r for the identifier of the call. The method call is evaluated to the value v in each of the principals in $\{\overline{p}\}$. Object-method calls block for the return value, and are unblocked once a quorum of principals make the same call, and the call is executed. The rule ObjReCall reduces an object-method call that is already executed (when the storage system is not compromised). It retrieves the return value from the recorded calls r. The rule ObjCallByz models the execution of an object-method call when the set of Byzantine principals is large enough to compromise the integrity of the storage system. In this case, the call returns an arbitrary value.

2.6 Information Flow Type System

In this section, we present the information flow type inference system and its guarantees. Instances of a well-typed class preserve their type specifications for confidentiality, integrity and availability at run time.

We present the type inference system in Figure 2.9. Given a class C, it yields constraints C on types and placements. The judgments are of the form $\Gamma, \mathcal{O}, \mathcal{M} \vdash C$, C

$$\textsc{ValT}\quad \Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash v : \bot,\ \emptyset$$

$$\textsc{VarT}\quad \Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash x : \Gamma(x),\ \emptyset$$

$$\textsc{OpT}\quad \frac{\begin{array}{c}\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e_i : \tau_i,\ C_i \quad \text{for } i \in \{1,2\} \\ \text{fresh } \tau \qquad C = C_1 \cup C_2 \cup \{\tau_1 \sqcup \tau_2 \sqsubseteq \tau\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e_1 \oplus e_2 : \tau,\ C}$$

$$\textsc{SeqT}\quad \frac{\begin{array}{c}\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e_1 : \tau_1,\ C_1 \quad \tau_{\mathsf{x}} = \langle c_{\mathsf{x}}, i_{\mathsf{x}}, a_{\mathsf{x}}\rangle \quad \tau_1 = \langle _, _, a_1\rangle \\ \Gamma[x \mapsto \tau_1], \mathcal{O}, \mathcal{M}, \mathcal{H}, \langle c_{\mathsf{x}}, i_{\mathsf{x}}, a_{\mathsf{x}} \sqcup a_1\rangle \vdash e_2 : \tau_2,\ C_2 \\ \text{fresh } \tau \qquad C = C_1 \cup C_2 \cup \{\tau_1 \sqcup \tau_2 \sqsubseteq \tau\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash x \coloneqq e_1; e_2 : \tau,\ C}$$

$$\textsc{IfT}\quad \frac{\begin{array}{c}\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e_0 : \tau_0,\ C_0 \\ \Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \sqcup \tau_0 \vdash e_i : \tau_i,\ C_i \quad \text{for } i \in \{1,2\} \qquad \text{fresh}\, \tau \qquad C = C_0 \cup C_1 \cup C_2 \cup \{\tau_0 \sqcup \tau_1 \sqcup \tau_2 \sqsubseteq \tau\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau,\ C}$$

$$\textsc{ThisCallT}\quad \frac{\begin{array}{c}\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e : \tau,\ C \quad \Gamma(m) = \tau'_{\mathsf{x}}, \tau_1 \to \tau_2 \quad \tau_1 = \langle _, _, a_1\rangle \\ \mathcal{M}(m) = \langle _, Q\rangle \qquad C' = C \cup \{\tau_{\mathsf{x}} \sqsubseteq \tau'_{\mathsf{x}}, \\ \tau \sqcup \tau_{\mathsf{x}} \sqsubseteq \tau_1,\ \ \mathit{Availability}(Q, \mathcal{H}) \sqsubseteq a_1\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash m(e) : \tau_2,\ C'}$$

$$\textsc{ObjCallT}\quad \frac{\begin{array}{c}\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e : \tau,\ C \quad \Gamma(o.m) = \tau_1 \to \tau_2 \quad \tau_1 = \langle _, _, a_1\rangle \\ \tau_2 = \langle c_2, _, _\rangle \qquad \mathcal{O}(o) = \langle _, _, Q\rangle \qquad C' = C \cup \{c_2 \sqsubseteq \mathcal{H}, \\ \tau \sqcup \tau_{\mathsf{x}} \sqsubseteq \tau_1,\ \ \mathit{Availability}(Q, \mathcal{H}) \sqsubseteq a_1\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash o.m(e) : \tau_2,\ C'}$$

$$\textsc{MethodT}\quad \frac{\begin{array}{c}\mathcal{M}(m) = \langle H, Q_c\rangle \quad \Gamma(m) = \tau_{\mathsf{x}}, \tau_1 \to \tau_2 \quad \tau_{\mathsf{x}} = \langle c_{\mathsf{x}}, _, _\rangle \quad \tau_1 = \langle c_1, i_1, _\rangle \quad \Gamma[x \mapsto \tau_1], \mathcal{O}, \mathcal{M}, H, \tau_{\mathsf{x}} \vdash e : \tau,\ C \\ C' = C \cup \{\tau \sqsubseteq \tau_2,\ \ \tau_1 \sqsubseteq \tau_2,\ \ c_1 \sqcup c_{\mathsf{x}} \sqsubseteq H,\ \ \mathit{CIntegrity}(Q_c) \sqsubseteq i_1\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M} \vdash m(x) \coloneqq e,\ C'}$$

$$\textsc{FieldT}\quad \frac{\begin{array}{c}\mathcal{O}(o) = \langle H, Q_s, Q_c\rangle \quad \Gamma(o) = \langle c, i, a\rangle \quad M(o) = \overline{m} \\ \overline{\Gamma(o.m) = \langle c_m, i_m, a_m\rangle \to \langle c'_m, i'_m, a'_m\rangle} \\ \overline{C_m = \{\langle c, i, a\rangle \sqsubseteq \langle c_m, i_m, a_m\rangle \sqsubseteq \langle c'_m, i'_m, a'_m\rangle\}} \quad C = \overline{\cup C_m} \cup \{\overline{\sqcup c'_m} \sqsubseteq H \ \ \mathit{SIntegrity}(Q_s) \sqsubseteq \overline{\sqcap i'_m}, \\ \mathit{Availability}(Q_s, H) \sqsubseteq \overline{\sqcap a'_m},\ \ \mathit{CIntegrity}(Q_c) \sqsubseteq \overline{\sqcap i_m}\}\end{array}}{\Gamma, \mathcal{O}, \mathcal{M} \vdash o,\ C}$$

$$\textsc{ClassT}\quad \frac{\overline{\Gamma, \mathcal{O}, \mathcal{M} \vdash o,\ C} \qquad \overline{\Gamma, \mathcal{O}, \mathcal{M} \vdash m,\ C'}}{\Gamma, \mathcal{O}, \mathcal{M} \vdash \langle \overline{o}, \overline{m}\rangle,\ \overline{\cup C} \cup \overline{\cup C'}}$$

Figure 2.9: Information Flow Type Inference System

that states that the class C is well-typed under the type environment Γ, and the placements $\mathcal{O}$ and $\mathcal{M}$ for the field objects and methods of C, if the constraints C are satisfied. (The judgments of the corresponding type *checking* system is $\Gamma, \mathcal{O}, \mathcal{M} \vdash C$. Instead of yielding constraints, it would simply check the same conditions.) The typing judgments for an object field o and a method definition d are $\Gamma, \mathcal{O}, \mathcal{M} \vdash o,\ C$ and $\Gamma, \mathcal{O}, \mathcal{M} \vdash d,\ C$ respectively. The typing judgments for an expression e is $\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e,\ C$ where $\mathcal{H}$ is the set of hosts that replicate the execution of e, and τ_{x} is the type of the context under which e is

executed. The context type $\tau_{\mathsf{x}} = \langle c_{\mathsf{x}}, i_{\mathsf{x}}, a_{\mathsf{x}} \rangle$ captures the implicit information flow. The context confidentiality type c_{x} represents the information that can be learned from the fact that the execution has reached the current expression e. Similarly, the context integrity type i_{x} represents the integrity of the information that determines the control flow to the current expression e. The conditions of the if expressions that enclose e determine its context confidentiality and integrity types. The context availability type a_{x} represents the availability of the information that the control flow requires to reach the current expression e. The conditions of the if expressions that enclose e, and the sequence expressions that precede e determine its context availability type.

The type environment Γ is a mapping from variables x and objects o to types τ, from this-methods m to function types $\tau_{\mathsf{x}}, \tau \to \tau'$, and from object-methods $o.m$ to function types $\tau \to \tau'$, where τ_{x} is the context type, τ is the parameter type, and τ' is the return type. The interfaces that objects expose can be called from any context. Thus, the type of their context parameter is $\top$ and is elided in an object interface.

Values, Variables, Operations, and Sequences. The rule ValT simply type-checks a value v as $\bot$, and the rule VarT type-checks a variable x according to the environment Γ. The rule OpT type-checks an operation $e_1 \oplus e_2$ as (a super-type of) the join of the types of the operands. Similarly, The rule SeqT type-checks a sequence $x \coloneqq e_1;\ e_2$ as the join of the types of the two operands. However, to type-check e_2, the type environment maps x to the type of e_1, and the availability of the context is reduced by the availability of e_1. The intuition is that e_2 cannot be evaluated if e_1 is unavailable.

Conditionals. The rule IfT type-checks a conditional expression as the join of the types of the condition and the branch expressions. To type-check the branch expressions, the given context type τ_{x} is joined with the type τ_0 of the condition expression e_0. The intuition is that e_0 implicitly flows to the branches. The fact that a branch is executed can leak the value of e_0. Therefore, the context confidentiality is increased by the confidentiality of τ_0. Further, the choice of the right branch is dependent on the integrity of e_0. Therefore, the context integrity is reduced by the integrity of τ_0. Further, the branches cannot be evaluated if e_0 is unavailable. Therefore, the context availability is reduced by the availability of τ_0.

Method Calls. The rule ThisCallT type-checks a this-method call $m(e)$. It first type-checks the argument e as τ, and then retrieves the type $\tau'_{\mathsf{x}}, \tau_1 \rightarrow \tau_2$ of m from the environment Γ, where τ'_{x} is the context parameter type, $\tau_1 = \langle _, _, a_1 \rangle$ is the parameter type, and τ_2 is the return type of the method. It then checks that the current context type τ_{x} is a subtype of the context parameter type τ'_{x}. It also checks that the argument type τ is a subtype of the parameter type τ_1. In addition, it checks that the current context type τ_{x} is a subtype of the parameter type τ_1. The intuition is that the argument can implicitly flow confidential information from the context (e.g. the enclosing conditionals) to the callee, and the integrity and availability of the context can affect the integrity and availability of the argument. Finally, the rule checks that the current hosts $\mathcal{H}$ that send the argument for the call to m meet the availability specification a_1 of the parameter. More precisely, let Q be the communication quorum system that an argument for a call to m is accepted from. If Q is restricted to $\mathcal{H}$, then it should be more available than a_1.

The rule ObjCallT type-checks an object-method call $o.m(e)$. It is similar in structure to ThisCallT with two differences. First, it does not include the constraint on context types since the context parameter type of an object-method is implicitly $\top$. Second, the hosts $\mathcal{H}$ should be confidential enough to observe the return value.

Methods. The rule MethodT type-checks a method definition $m(x) \coloneqq e$. It first retrieves the type $\tau_{\mathsf{x}}, \tau_1 \rightarrow \tau_2$ of m from the given environment Γ. It then type-checks the body e under the context type τ_{x}, and the environment Γ extended with x typed as τ_1. The resulting type τ of e has to be a subtype of the return type τ_2. Further, the parameter type τ_1 has to be a subtype of the return type τ_2. This makes a this-method call have a stronger type than its argument. Further, the hosting principals H should be more confidential than c_1 and c_{x} because the hosts can learn confidential information in the argument and about the context from the fact that the call is made. Finally, the communication quorum system that accepts the arguments for m should provide more integrity than the integrity i_1 that the parameter expects.

Field Objects. The rule FieldT checks the following conditions for an object o. (1) Let $\langle c, i, a \rangle$ be the type of o in the environment Γ. The rule checks that c is a lower bound for the confidentiality, and i and a are higher bounds for the integrity and availability of the parameters and return values of the methods m of o. (These bounds are used to state the non-interference properties in the next section.) (2) Let H be the hosts for the the storage quorum system Q_s of o. (2.1) The rule checks that the hosting principals H are confidential enough to host the methods of o. More precisely, it checks that the join (i.e., intersection) of the confidentiality of the return values c'_m can flow to (i.e., is a superset of)

H. (2.2) Further, in order to host o, Q_s should provide more integrity and availability than the integrity i'_m and availability a'_m that the return value of each method m is expected to have. (3) Let Q_c be the communication quorum system that the arguments of method calls on o are accepted from. Q_c should provide more integrity than the integrity i_m that the parameter of each method m expects.

Class. The rule ClassT type-checks a class $\langle \overline{o}, \overline{m} \rangle$ by type-checking each object o and method m.

2.7 Security and Resiliency Guarantees

The type system guarantees non-interference for confidentiality, integrity and availability of methods of well-typed classes. Further, their integrity and availability types characterize their resilience to Byzantine attacks. In this section, we will see that if a method is typed, it enjoys non-interference from objects of super-types, and resilience to Byzantine attacks of sub-types. After basic definitions, we first look at the non-interference theorems and then the resilience theorems. The proofs are available in the appendix section 2.13.

The partitioning process splits methods to a sequence of methods. In the initial state, the client invokes the first method in that sequence that we call the initial method.

Definition 1 (Initial method) *The initial (or client) method m_0 is a method that is hosted on a non-Byzantine (client) principal p_0 and can be directly invoked by p_0. More precisely, if $\mathcal{B}$ denotes the set of Byzantine principals, and $\mathcal{M}$ denotes the method placement, then $p_0 \notin \mathcal{B}$ and $\mathcal{M}(m_0) = \langle \{p_0\}, \{\{p_0\}\} \rangle$.*

Definition 2 (Initial distributed state) *In the initial distributed state, p_0 calls m_0 with the initial argument v_0, the objects have their initial states, and the set of messages in the system is empty. More precisely, the initial distributed state is $\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle$ where $\mathbb{P}_0 = [\overline{p \mapsto \langle \bot, 0, 0 \rangle}][p_0 \mapsto \langle m_0(v_0), 0, 0 \rangle]$, $\mathbb{S}_0 = [\overline{o \mapsto \langle S_0(o), \emptyset \rangle}]$, where $S_0(o)$ is the initial state of o.*

To state properties over only a subset of objects in the maps $\mathcal{O}$ and $\mathbb{S}$, we project them over the three type kinds.

Definition 3 (Projection over types) *Given a typing environment Γ, and a map M on the objects domain, the projection of M over a confidentiality c restricts the domain of M to objects with confidentiality* less than *c in Γ. More precisely, $M \,|_\Gamma\, c = M \,|\, \{o \mid \mathsf{let}\ \langle c', _, _ \rangle := \Gamma(o)\ \mathsf{in}\ c' \sqsubseteq c\}$. Similarly, the projection of M over an integrity i restricts the domain of m to objects with integrity* more than *i. The projection over an availability a is similarly to objects of more availability than a. More, precisely, $M \,|_\Gamma\, i = M \,|\, \{o \mid \mathsf{let}\ \langle _, i', _ \rangle := \Gamma(o)\ \mathsf{in}\ i' \sqsubseteq i\}$ and $M \,|_\Gamma\, a = M \,|\, \{o \mid \mathsf{let}\ \langle _, _, a' \rangle := \Gamma(o)\ \mathsf{in}\ a' \sqsubseteq a\}$.*

We lift the integrity of storage quorum systems, *SIntegrity*, to object placements $\mathcal{O}$, as the join of *SIntegrity* of the storage systems of all the objects in $\mathcal{O}$. More precisely, $\mathit{SIntegrity}([\overline{o \mapsto \langle _, Q, _ \rangle}]) = \overline{\sqcup\, \mathit{SIntegrity}(Q)}$

Non-Interference. The type that the type system associates with a method m captures the trustworthiness of the objects that it accesses. If the return type of m is τ, then m accesses only objects that are typed as sub-types of τ, and enjoys non-interference from objects that are typed as super-types of τ. In fact, there is non-interference even if the super-type relation holds only for one of the three type components. If the type system associates

a confidentiality type with (the return value of) m, then calls to m don't access *objects of higher confidentiality*. Similarly, if the type system associates an integrity or availability type with m, then calls to m don't access *objects of lower integrity or availability*. Changing the state of these objects doesn't interfere with the return value.

Confidentiality. Assume that the client method is type-checked with the confidentiality type c. Let O_c be the objects that are less (or as) confidential than c. The following non-interference theorem states that if two state maps $\mathbb{S}_1$ and $\mathbb{S}_2$ have the same states for the objects O_c, and the integrity of O_c is not compromised by the Byzantine principals $\mathcal{B}$, then any two executions with $\mathbb{S}_1$ and $\mathbb{S}_2$ that return a value to the client, return the same value. The integrity of the objects is required since a compromised object that loses integrity can behave non-deterministically.

Theorem 4 (Non-interference) *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, C, $\mathcal{B}$, c, i, a, $\mathbb{S}_1$, $\mathbb{S}_2$, $\mathbb{P}'_1$, $\mathbb{P}'_2$, L, v *and* v', *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$, *and* $\Gamma(m_0) = _, _ \rightarrow \langle c, i, a \rangle$, *and either*

- $\mathbb{S}_1 \,|_\Gamma\, c = \mathbb{S}_2 \,|_\Gamma\, c$, *and* $\mathit{SIntegrity}(\mathcal{O} \,|_\Gamma\, c) \sqsubseteq \{\mathcal{B}\}$, *or*
- $\mathbb{S}_1 \,|_\Gamma\, i = \mathbb{S}_2 \,|_\Gamma\, i$, *and* $\mathit{SIntegrity}(\mathcal{O} \,|_\Gamma\, i) \sqsubseteq \{\mathcal{B}\}$, *or*
- $\mathbb{S}_1 \,|_\Gamma\, a = \mathbb{S}_2 \,|_\Gamma\, a$, *and* $\mathit{SIntegrity}(\mathcal{O} \,|_\Gamma\, a) \sqsubseteq \{\mathcal{B}\}$, *or*

$\langle \mathbb{P}_0, \mathbb{S}_1, \emptyset \rangle \overset{L}{\rightsquigarrow}{}^{*} \langle \mathbb{P}'_1, _, _ \rangle$, *and* $\langle \mathbb{P}_0, \mathbb{S}_2, \emptyset \rangle \overset{L}{\rightsquigarrow}{}^{*} \langle \mathbb{P}'_2, _, _ \rangle$,

$\mathbb{P}'_1(p_0) = \langle v, _, _ \rangle$, *and* $\mathbb{P}'_2(p_0) = \langle v', _, _ \rangle$,

then $v = v'$.

The proof is by induction on the steps. For confidentiality, every step preserves the invariant that both the expressions in non-Byzantine principals, and the requested this-method calls in messages are less confidential than c. Thus, only objects that are less confidential than c are accessed. Further, these objects are assumed to have the same state in $\mathbb{S}_1$ and $\mathbb{S}_2$, and preserve integrity. Therefore, object-method calls behave deterministically and return the same value in the two executions, which in turn preserves the equality of the expression of every non-Byzantine principal in the two executions.

Integrity. Assume that the client method is type-checked with the integrity type i. Let O_i be the objects with integrity more than i. The above non-interference theorem states that if two state maps $\mathbb{S}_1$ and $\mathbb{S}_2$ have the same states for the objects O_i, and the integrity of O_i is not compromised by the Byzantine principals $\mathcal{B}$, then any two executions with $\mathbb{S}_1$ and $\mathbb{S}_2$ that return a value to the client, return the same value.

Availability. Similarly, the above theorem states non-interference for availability. If a method is type-checked with the availability type a, then different states for objects that are less available than a cannot interfere with the return value.

Resilience. Well-typed classes are resilient to Byzantine principals. In particular, the integrity and availability types that the type system associates with a method characterize the resilience of its integrity and availability to Byzantine principals. The integrity of the method is resilient to any Byzantine attack that is subsumed by the integrity type of (the return value of) the method. Similarly, the availability of the method is resilient to any Byzantine attack that is subsumed by its integrity and availability types.

Integrity Resiliency. If the *Byzantine principals are subsumed by the integrity type*, then *the results of distributed executions is the same value as the sequential execution.* More precisely, if the sequential semantics evaluates the client method call to the value v, and the set of Byzantine principals $\mathcal{B}$ is subsumed by the integrity type i of (the return value of) the method, then any distributed execution of the method call that results in a value, results in v as well. For example, in Figure 2.1.(a) and (b), the integrity of the return type τ of *transfer* is $\mathcal{P}_2(A) \times_{\cup} \mathcal{P}_1(B)$. Therefore, the result of a distributed execution of *transfer* is the same as its sequential execution even if two A and one B principals are Byzantine.

Theorem 5 (Integrity Resilience) *For all* v, Γ, $\mathcal{O}$, $\mathcal{M}$, C, i, $\mathcal{B}$, $\mathbb{P}$, *and* v', *if*

$$\langle m_0(v_0), S_0\rangle \rightarrow^* \langle v, _\rangle,$$

$$\Gamma, \mathcal{O}, \mathcal{M} \vdash C, \quad \Gamma(m_0) = _, _ \rightarrow \langle _, i, _\rangle, \quad \textit{and} \quad i \sqsubseteq \{\mathcal{B}\},$$

$$\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset\rangle \rightsquigarrow^* \langle \mathbb{P}, _, _\rangle, \quad \textit{and} \quad \mathbb{P}(p_0) = \langle v', _, _\rangle, \; v' \neq \bot,$$

then $v' = v$.

The proof is by induction on macro-steps where a step is taken by all replicating principals before the next step is taken. For every execution, there is a corresponding macro-step execution. A macro-step preserves the invariant that the expression of every non-Byzantine principal has more integrity than i, and there is a sequential execution from the initial call to that expression. In the case for execution of a this-method call, we show that the request is received from at least one non-Byzantine principal and use the invariant for that principal.

Availability Resiliency. If the *Byzantine principals are subsumed by the integrity and availability types*, then a *distributed execution can make progress and results in the same*

value as the sequential execution. More precisely, if the sequential semantics evaluates the client method call to the value v, and the set of Byzantine principals $\mathcal{B}$ is subsumed by the integrity i and availability a types of (the return value of) the method, then a distributed execution results in v as well. For example, in Figure 2.1.(a) and (b), the integrity and availability of the return type τ of *transfer* are $\mathcal{P}_2(A) \times_\cup \mathcal{P}_1(B)$, and $\mathcal{P}_1(A) \times_\cup \mathcal{P}_1(B)$ respectively. Therefore, a distributed execution of *transfer* results in the same value as its sequential execution, even if one A and one B principals are Byzantine.

Theorem 6 (Availability Resilience) *For all v, Γ, $\mathcal{O}$, $\mathcal{M}$, C, i, a, $\mathcal{B}$, and $\mathbb{P}$, if*

$$\langle m_0(v_0), S_0 \rangle \rightarrow^* \langle v, _ \rangle,$$

$$\Gamma, \mathcal{O}, \mathcal{M} \vdash C, \quad \Gamma(m_0) = _, _ \rightarrow \langle _, i, a \rangle, \quad \textit{and} \quad i \sqcup a \sqsubseteq \{\mathcal{B}\},$$

then there exists $\mathbb{P}$ such that

$$\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle \rightsquigarrow^* \langle \mathbb{P}, _, _ \rangle, \quad \textit{and} \quad \mathbb{P}(p_0) = \langle v, _, _ \rangle.$$

The set of Byzantine principals should be subsumed by not only the availability type a but also the integrity type i so that the storage quorum systems of the objects keep their quorum intersection and return sound values. Compromised return values can drift the execution to paths that do not match the sequential execution, and can even lead to non-termination.

We note that while Theorem 4 states the safety property that a typed method does not access objects that are less available than its type, Theorem 6 states the liveness property that it makes progress despite Byzantine principals that are not as strong as its integrity and availability types.

2.8 Constraint Solving

We translate the constraints C to the theory of linear arithmetic. Let the number of principal classes (trust domains) be n and let P_j denote the set of principals of class j. For example, in Figure 2.1, n is 3, and the principal classes P_0, P_1 and P_2 are A, B and $C = \{p_0\}$. We represent a confidentiality value c as a tuple $\langle c_0, c_1, .., c_{n-1} \rangle$ where each c_j represents an integer variable with the value 1 or 0. The principal class j is trusted or untrusted if the value of c_j is 1 or 0 respectively.

We represent a set of hosting principals H as subsets of given sizes of the principal classes. The hosting principals H are represented as a tuple where H_j represents the number of hosting principal from the class j. Similarly, we represent a quorum system Q or a resiliency value B as subsets of certain sizes of the principal classes. For example, in Figure 2.1.(a), the availability of τ is $\mathcal{P}_2(A) \times_\cup \mathcal{P}_1(B)$ that is all the subsets with 2 principals from A and 1 principals from B. This can be succinctly represented as the tuple $\langle 2, 1, 0 \rangle$. In general, we represent the set of subsets $s = \mathcal{P}_{s_0}(P_0) \times_\cup \mathcal{P}_{s_1}(P_1) \times_\cup .. \times_\cup \mathcal{P}_{s_{n-1}}(P_{n-1})$ as $\langle s_0, s_1, .., s_{n-1} \rangle$. We let a quorum system Q (or resiliency value B) be the union of n such sets: $Q = \cup_{i \in \{0,1,..,n-1\}} \langle Q_{ij_0}, Q_{ij_1}, .., Q_{ij_{n-1}} \rangle$. The indices i range over n tuples. In the tuple with index i, the number of principals from class j is Q_{ij}. Having n such tuples keeps the space of quorums tractable, and is expressive enough to capture common quorum systems. For example, the quorum system $\mathcal{P}_4(P_0) \cup \mathcal{P}_5(P_1) \cup \mathcal{P}_2(P_2)$ is represented as $\langle 4, 0, 0 \rangle \cup \langle 0, 5, 0 \rangle \cup \langle 0, 0, 2 \rangle$ that is all the subsets of size 4 from P_0, of size 5 from P_1, and of size 2 from P_2. Thus, each quorum system Q or resiliency value B can be represented by n^2 variables.

We now define the translation of two constraints. (The translation of all the constraints is available in the appendix.) Each constraint is translated to a constraint of size at most $O(n^4)$.

$$\mathit{CIntegrity}(Q) \sqsubseteq B \triangleright \bigwedge_i \bigwedge_{i'} \bigvee_j Q_{ij} > B_{i'j}$$

$$\mathit{Availability}(Q, H) \sqsubseteq B \triangleright \bigwedge_{i'} \bigvee_i \bigwedge_j Q_{ij} > 0 \rightarrow Q_{ij} \leq H_j - B_{i'j}$$

In the first rule, the assertion states that none of the quorums of Q is contained in a Byzantine set of B. More precisely, for each tuple i in Q, and tuple i' in B, for at least one of the indices (principal classes) j, Q_{ij} is more than $B_{i'j}$. In the second rule, the assertion states that for every Byzantine set b in B, there is at least one quorum q in Q that falls inside the hosts H and doesn't intersect b. More precisely, for each tuple i' in B, there is a tuple i in Q such that for all indices j, if Q_{ij} is non-zero, it is less than or equal to H_j minus $B_{i'j}$.

We minimize the number principals in the host sets and quorum systems to reduce the load. Let $\mathcal{M} = [\overline{m \mapsto \langle H, Q \rangle}]$ and $\mathcal{O} = [\overline{o \mapsto \langle H', Q', Q'' \rangle}]$. The optimization constraint is $\mathsf{min}\ \overline{\sum_j H_j + H'_j} + \overline{\sum_i \sum_j Q_{ij} + Q'_{ij} + Q''_{ij}}$ that minimizes the size of hosts and quorums. For example, in our running example, the one-time transfer, for the given specification in Figure 2.1.(a), type inference can find the correct placement that we saw in Figure 2.1.(d). If we add the minimization constraint, the more efficient placement $\langle A_{1..5},\ \mathcal{P}_4(A_{1..5}),\ \mathcal{P}_3(A) \rangle$ for r_1 can be found; it has 5 hosts and quorums of size 4.

2.9 Implementation and Experiments

We developed a tool called Hamraz and experimented with multiple resiliency specifications for six use-cases on a cluster of nodes. The experiments show that Hamraz

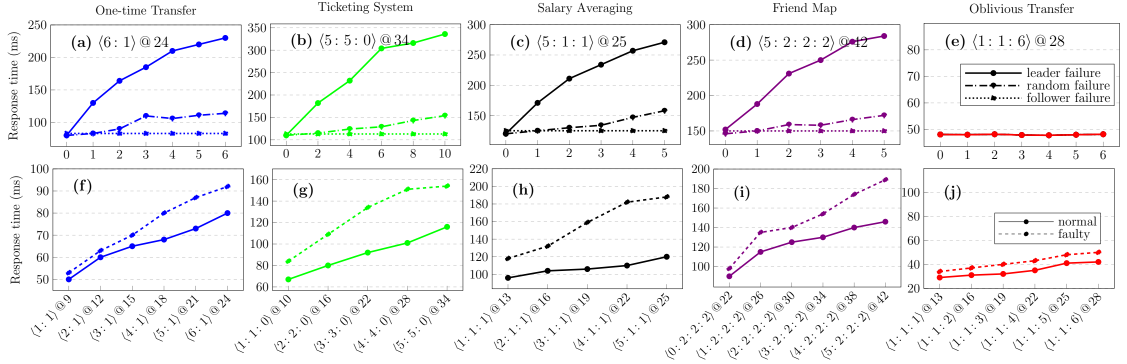


Figure 2.10: Top row: Response time for increased faults. Bottom row: Response time for increased resiliency. $\mathcal{P}_i(A) \times_{\cup} .. \times_{\cup} \mathcal{P}_k(Z)$ is denoted as $\langle i, .., k \rangle @ n$ where n is the total number of principals.

can successfully infer the placements and replications for the given resiliency specifications. Further, it generates systems that can gracefully tolerate injected faults that are as strong as the resiliency specifications, and it can adjust the level of replication according to the resiliency specifications. We make Hamraz publicly available as open-source software.

Implementation. Hamraz is implemented in Java in two parts: the synthesizer and the runtime. The synthesizer closely follows the partitioning and type inference system of section 2.4 and section 2.6, and the runtime closely follows the distributed operational semantics of section 2.5. We used the Z3 SMT solver (v. 4.8.10) [144] for constraint solving and optimization. We implemented communication quorum systems using SSL StartTLS from the Netty library [11], and storage quorum systems for field objects using the BFT-SMaRt library [66].

Platform. The experiments are done on a high-performance cluster with Intel Broadwell CPUs with 4 cores, 2GB of RAM and CentOS-7 Linux x86_64 3.10.0. JDK is OpenJDK RE 18.9. The runtime is executed for each principal on a separate node of the

	PT (s)	CN	GT (ms)	ST (s)	TT (s)
One-time Transfer	0.03	181	10.6	6.08	6.12
Ticket System	1.18	525	12.60	214.46	215.65
Oblivious Transfer	0.04	203	14.6	9.96	10.01
Auction	36.43	681	10.60	22.53	58.98
Friend Map	0.03	302	13.2	359.94	359.98
Salary Sharing	26.74	611	14.2	160.38	187.13

Table 2.1: Partitioning and Type Inference. PT: Partitioning time, CN: Constraints number, GT: Constraint generation time, ST: Constraint solving time, TT: Total time

cluster. The nodes are connected with 56 Gb/s InfiniBand. Each reported number is the arithmetic mean of 5 repetitions. Each repetition is the average of the response time for 150 method calls.

Use-cases. We experiment with six use-cases: one-time transfer that we saw in section 5.2, oblivious transfer [451], ticketing system, auction system, friend map [299] and privacy-preserving salary averaging [308].

Consider the sets of principals A to Z. In our plots, we concisely represent a resiliency specification $\mathcal{P}_i(A) \times_{\cup} .. \times_{\cup} \mathcal{P}_k(Z)$ as $\langle i, .., k \rangle \,@\, n$ where n is the total number of principals that is $(3 \times i + 1) + .. + (3 \times k + 1)$ plus 1 for the client. We use the same resiliency specification for both integrity and availability.

Partitioning and Type Inference. Hamraz successfully partitioned and inferred the placement and replication for each field object and method of the above use-cases. Table 2.1 shows the detailed execution times. The process takes less than 6 minutes. We see that the process is often dominated by constraint solving. However, when the use-case has a large number of object-method calls (e.g. the Auction use-case), the CPS transformation takes longer and the partitioning time is a larger faction.

Increasing Faults. In this experiment, we validate the hypothesis that HAMRAZ generates replicated systems that are as resilient as the specifications require. In this experiment (the first row of Figure 2.10), the specification is fixed and is written in the plot of each use-case. We consider the effect of increasing the number of injected faults on the response time. Injected faults are randomly selected from three types: crash fault, corrupted payload, and delayed response. In these plots, we increase the number of faults for the principals (a) A, (b) A and B at the same time, (c) A, (d) A, and (e) C, from 0 to their specified resiliency. In plots (a)-(d), the failing principals host objects but in plot (e), the failing principals host only methods. Each plot shows three lines for three fault injection scenarios: solid: failing only leaders, dotted: failing only followers, and dashed: failing randomly.

In all the failure scenarios across the use-cases, the resulting systems can gracefully tolerate the injected faults. Random failures increase the response time between 3 and 42 percent. Further, we observe that tolerating failing leaders is considerably slower than failing followers due to leader reconfiguration. We observe in plots (a)-(d) that increasing the faults increases the response time. On the other hand, in plot (e), the increase in response time is negligible (overlapping lines). This is due to the fact that in contrast to principals that host objects (plots (a)-(d)), tolerating failure of principals that host methods (plot (e)) does not require reconfiguration.

Increasing Resiliency. In the following two experiments, we validate the hypothesis that HAMRAZ can adjust replication according to the strength of the specification. In the second row of Figure 2.10, we consider the effect of increasing resiliency on the

response time in normal (solid line) and faulty executions (dashed line) with maximum number of faults. Higher resiliency requires more replicas and more communication between them that affects the overall response time. If a fixed number of replicas was used, the response time would be flat.

Increasing the Load. In the auction use-case, if the initial offer o is less than 350, then A immediately wins with the offer $o - 1$. Otherwise, the two agents beat each other's offers $o - 350$ times until A offers 349 and wins. Figure 2.11 shows the response time for increased initial offers. Larger initial offers lead to more object-calls and increased load. We experimented with three specifications where the resiliency for A and B is doubled from one to the next. We observe that as the load increases, the response time of more resilient systems grows slightly faster. More resilient system require more principals and more coordination.

2.10 Related Works

Information flow control [147, 393, 374, 426, 347, 45, 59, 112, 364, 242] has been widely used to enforce confidentiality and integrity policies. It has been applied to concurrent [406] and distributed [392] programs on trusted hosts. Further, Fabric [299] supports programming distributed systems on heterogeneously trusted hosts, and enforces confidentiality and integrity types, but doesn't provide Byzantine replication and doesn't enforce availability policies. Several previous works [92, 356, 424, 129, 125, 126, 352] automatically partition applications to multiple tiers, often to the web server and client tiers, and enforce confidentiality and integrity, but not availability. Jif/split [451, 455] partitions programs

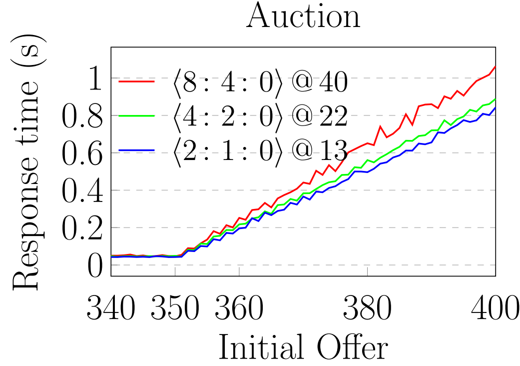


Figure 2.11: Response time for increased load

and replicates code partitions and data. It can replicate commitments instead of cleartexts to increase integrity without reducing confidentiality. Further, its secure communication assumptions between partitions were later lifted by a cryptographic back-end [181]. PtrSplit [301] splits programs with C++ pointers. However, these projects do not provide availability; in fact, Jif/split may reduce availability as all replicas need to be available. Another work [46] synthesizes cryptographic implementations for distributed applications; however, it does not consider availability polices.

Later, information flow type systems were applied to enforce availability policies [456] but assumed availability of the computation platform and did not consider Byzantine-resilient replication and their type lattices. Similarly, RMS [298] adjusts the placement and replication of objects based on availability and performance specifications; however, it does not tolerate Byzantine failures. Qimp [457] provides a language construct for clients to run an expression on references at a use-specified quorum, and type-checks availability guarantees. (In addition, when it is provable that integrity is compromised, it can use a default value to provide low integrity but high availability.) In contrast, this

paper allows the user to describe a class composed of field objects and methods, with confidentiality, integrity and particularly availability type policies, and without distribution details. It then automatically partitions the class and infers adequate Byzantine quorum systems for methods and field objects to enforce the three polices.

State-machine replication is a well-known technique often used to tolerate crash failures [261, 99, 359, 358]. Byzantine failures were coined in the Byzantine Generals agreement problem [268] together with a few early protocols for Byzantine replication. Later, the more practical PBFT protocol [114], and an abundant number of optimized variants such as Q/U [13], HQ [140], Zyzzyva [249], Stewart [30], ABSTRACT [42], MinBFT [422], CheapBFT [238], ZZ [436], UpRight [131], BFT2F [286], Aardvark [133] and HoneyBadgerBFT [337] appeared. Further, researchers verified the replication protocols [362, 152, 376, 241, 231, 119, 447, 186, 217, 377, 292]. However, these projects only consider a monolithic replicated system. In contrast, this project supports classes whose methods are implemented based on multiple objects. It partitions each method and separately replicates each partitioned method and field object, and yet guarantees end-to-end non-interference and resiliency.

2.11 Conclusion

This paper presented a theoretical framework and a system for trustworthy distributed systems. It includes a lattice model of resiliency, a security-typed object-based language to capture end-to-end type polices for the three aspects of trustworthiness, a partitioning transformation, operational semantics, an information flow type inference sys-

tem, and quorum constraint solving to automatically construct partitioned and replicated systems that guarantee non-interference and resiliency properties especially for availability.

2.12 Constraint Solving

Core Constraints. The constraints C that the type system generates are of the following forms.

$$
\begin{array}{rlr}
C & := & \text{Constraint} \\
& \mid\ \tau \sqsubseteq \tau & C_1 \\
& \mid\ \tau \sqcup \tau \sqsubseteq \tau & C_2 \\
& \mid\ \tau \sqcup \tau \sqcup \tau \sqsubseteq \tau & C_3 \\
& \mid\ c \sqsubseteq c' & C_4 \\
& \mid\ i \sqsubseteq i' & C_5 \\
& \mid\ a \sqsubseteq a' & C_6 \\
& \mid\ \overline{\sqcup c'_m} \sqsubseteq \cup Q & C_7 \\
& \mid\ \mathit{CIntegrity}(Q) \sqsubseteq \overline{\sqcap i_m} & C_8 \\
& \mid\ \mathit{SIntegrity}(Q) \sqsubseteq \overline{\sqcap i'_m} & C_9 \\
& \mid\ \mathit{Availability}(Q, H) \sqsubseteq \overline{\sqcap a'_m} & C_{10} \\
& \mid\ \mathit{Availability}(Q, H) \sqsubseteq a & C_{11}
\end{array}
$$

The constraints C can be reduced to the core constraints $\mathcal{A}$ below. By distribution of $\sqcup$ over $\sqsubseteq$ and then decomposing types to their three elements, the constraints C_1, C_2 and C_3 is reduced to $\mathcal{A}_1$ and $\mathcal{A}_2$. The constraint C_4 is $\mathcal{A}_1$. The constraint C_5 and C_6 are $\mathcal{A}_2$. By distribution of $\sqcup$ over $\sqsubseteq$, the constraint C_7, is reduced to $\mathcal{A}_3$. By distribution of $\sqsubseteq$ over $\sqcap$, the constraints C_8, C_9 and C_{10} are reduced to $\mathcal{A}_4$, $\mathcal{A}_5$ and $\mathcal{A}_6$ respectively.

$$
\begin{array}{rlr}
\mathcal{A} & := & \text{Core Constraint} \\
& \mid \quad c \sqsubseteq c' & \mathcal{A}_1 \\
& \mid \quad B \sqsubseteq B' & \mathcal{A}_2 \\
& \mid \quad c \sqsubseteq \cup Q & \mathcal{A}_3 \\
& \mid \quad \mathit{CIntegrity}(Q) \sqsubseteq B & \mathcal{A}_4 \\
& \mid \quad \mathit{SIntegrity}(Q) \sqsubseteq B & \mathcal{A}_5 \\
& \mid \quad \mathit{Availability}(Q, H) \sqsubseteq B & \mathcal{A}_6
\end{array}
$$

Translation. We translate the constraints C to the theory of linear arithmetic. Let the number of principal classes (trust domains) be n and let P_j denote the set of principals of class j. For example, in Figure 2.1, n is 3, and the principal classes P_0, P_1 and P_2 are A, B and $C = \{p_0\}$. We represent a confidentiality value c as a tuple $\langle c_0, c_1, .., c_{n-1} \rangle$ where each c_j represents an integer variable with the value 1 or 0. The principal class j is trusted or untrusted if the value of c_j is 1 or 0 respectively.

We represent a set of hosting principals H as subsets of given sizes of the principal classes. The hosting principals H are represented as a tuple where H_j represents the number of hosting principal from the class j. Similarly, we represent a quorum system Q or a resiliency value B as subsets of certain sizes of the principal classes. For example, in Figure 2.1.(a), the availability of τ is $\mathcal{P}_2(A) \times_{\cup} \mathcal{P}_1(B)$ that is all the subsets with 2 principals from A and 1 principals from B. This can be succinctly represented as the tuple $\langle 2, 1, 0 \rangle$. In general, we represent the set of subsets $s = \mathcal{P}_{s_0}(P_0) \times_{\cup} \mathcal{P}_{s_1}(P_1) \times_{\cup} .. \times_{\cup} \mathcal{P}_{s_{n-1}}(P_{n-1})$ as $\langle s_0, s_1, .., s_{n-1} \rangle$. We let a quorum system Q (or resiliency value B) be the union of n such sets: $Q = \cup_{i \in \{0,1,..,n-1\}} \langle Q_{ij_0}, Q_{ij_1}, .., Q_{ij_{n-1}} \rangle$. The indices i range over n tuples. In

the tuple with index i, the number of principals from class j is Q_{ij}. Having n such tuples keeps the space of quorums tractable, and is expressive enough to capture common quorum systems. For example, the quorum system $\mathcal{P}_4(P_0) \cup \mathcal{P}_5(P_1) \cup \mathcal{P}_2(P_2)$ is represented as $\langle 4, 0, 0 \rangle \cup \langle 0, 5, 0 \rangle \cup \langle 0, 0, 2 \rangle$ that is all the subsets of size 4 from P_0, of size 5 from P_1, and of size 2 from P_2. Thus, each quorum system Q or resiliency value B can be represented by n^2 variables. We note that duplicating a tuple does not change the represented set.

We now define the translation of constraints to the theory of linear arithmetic. First, we generate the following constraints for each variable c, H, Q, and B.

$$
\begin{array}{rcl}
\text{Variable } c & \triangleright & \bigvee_j \; c_j = 0 \vee c_j = 1 \\
\text{Variable } H & \triangleright & \bigwedge_j \; 0 \leq H_j \leq |P_j| \\
\text{Variable } Q & \triangleright & \bigwedge_i \bigwedge_j \; 0 \leq Q_{ij} \leq |P_j| \\
\text{Variable } B & \triangleright & \bigwedge_i \bigwedge_j \; 0 \leq B_{ij} \leq |P_j|
\end{array}
$$

We now define the translation of each constraint $\mathcal{A}$. Each constraint is translated to a constraint of size at most $O(n^4)$.

$$c \sqsubseteq c' \quad \triangleright \quad \bigwedge_{j} c'_j = 1 \rightarrow c_j = 1 \qquad T_1$$

$$B \sqsubseteq B' \quad \triangleright \quad \bigwedge_{i'} \bigvee_{i} \bigwedge_{j} B'_{i'j} \leq B_{ij} \qquad T_2$$

$$c \sqsubseteq \cup Q \quad \triangleright \quad \bigwedge_{i} \bigwedge_{j} Q_{ij} > 0 \rightarrow c_j = 1 \qquad T_3$$

$$\mathit{CIntegrity}(Q) \sqsubseteq B \quad \triangleright \quad \bigwedge_{i} \bigwedge_{i'} \bigvee_{j} Q_{ij} > B_{i'j} \qquad T_4$$

$$\mathit{SIntegrity}(Q) \sqsubseteq B \ \text{ for } \ \mathcal{O}(o) = \langle H, Q, _ \rangle \quad \triangleright \qquad T_5$$

$$\bigwedge_{i_1} \bigwedge_{i_2} \bigwedge_{i'} \bigvee_{j} Q_{i_1 j} + Q_{i_2 j} - H_j > B_{i'j}$$

$$\mathit{Availability}(Q, H) \sqsubseteq B \quad \triangleright \qquad T_6$$

$$\bigwedge_{i'} \bigvee_{i} \bigwedge_{j} Q_{ij} > 0 \rightarrow Q_{ij} \leq H_j - B_{i'j}$$

The translation rule T_1 translates a flow relation between two confidentiality variables c and c'. The assertion states that the principal classes that are authorized by c' are authorized by c as well. More precisely, for each index j (principal class) of the tuples c and c', if c'_j is 1 then c_j is 1 as well.

The translation rule T_2 translates a flow relation between two resiliency variables B and B'. The assertion states that every set in B' is a subset of a set in B. More precisely, for each tuple i' in B', there is a tuple i in B such that for each index j (principal class), $B'_{i'j}$ is less than or equal to B_{ij}.

The translation rule T_3 translates the flow relation from a confidentiality variable c to the union of the quorums of a quorum system Q. The assertion states that if a quorum in Q has a principal in class j, then the class j is authorized by c as well. More precisely, if there is a tuple i in Q whose index j is greater than zero, then c_j is 1.

The translation rule T_4 translates the constraint that the integrity of the communication quorum system Q is stronger than a resiliency variable B. The assertion states that none of the quorums of Q is contained in a failure set of B. More precisely, for each tuple i in Q, and tuple i' in B, at least in one of the indices (principal classes) j, Q_{ij} is more than $B_{i'j}$.

The translation rule T_5 translates the constraint that the integrity of the storage quorum system Q is stronger than a resiliency variable B. The assertion states that the intersection of no two quorums of Q is contained in a failure set of B. More precisely, for each pair of tuples i_1 and i_2 in Q, and tuple i' in B, for at least one of the indices (principal classes) j, the sum of Q_{i_1j} and Q_{i_2j} minus the size of the principal class j in the hosting principals H is more than $B_{i'j}$.

The translation rule T_6 translates the constraint that the availability of the quorum system Q on the set of hosts H is stronger than a resiliency variable B. The assertion states that for every Byzantine set in B, there is at least one quorum in Q that falls inside the

hosts H and does not intersect the Byzantine set. More precisely, for each tuples i' in B, there is a tuple i in Q such that for every index (principal class) j, Q_{ij} is less than or equal to H_j minus $B_{i'j}$. The implication with the premise $Q_{ij} > 0$ requires the condition above for only the classes j that the quorums Q_i have a member from. Otherwise, when $Q_{ij} = 0$, no host in principal class j is needed but the condition fails if $H_j < B_{i'j}$.

Optimization. We want to satisfy the constraints above while minimizing the number of hosts. Replication on fewer hosts reduces the load on the system. Let $\mathcal{M} = [\overline{m \mapsto \langle H, Q \rangle}]$ and $\mathcal{O} = [\overline{o \mapsto \langle H', Q', Q'' \rangle}]$. The optimization constraint is

$$\mathsf{min} \; \overline{\sum_j H_j + H'_j} + \overline{\sum_i \sum_j Q_{ij} + Q'_{ij} + Q''_{ij}}$$

It minimizes the number of principals that host methods and objects, and the number of principals in the communication and storage quorum systems. Weighted factors can be added to the formula incorporate their relative load.

We note that duplicating a tuple in the representation of a quorum system does not change the represented quorum system. If there are duplicate tuples in the solution for a quorum system variable Q (or resiliency variable B), a post-process filters them and keeps only one instance of each tuple.

2.13 Security Guarantees

2.13.1 Confidentiality Non-Interference

Theorem 7 (Confidentiality Non-interference) *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, C, $\mathcal{B}$, c, $\mathbb{S}_1$, $\mathbb{S}_2$, $\mathbb{P}'_1$, $\mathbb{P}'_2$, L, v *and* v', *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C, \quad and \quad \Gamma(m_0) = _, _ \rightarrow \langle c, _, _\rangle,$

$\mathbb{S}_1 \,|_\Gamma\, c = \mathbb{S}_2 \,|_\Gamma\, c, \quad and \quad SIntegrity(\mathcal{O} \,|_\Gamma\, c) \sqsubseteq \{\mathcal{B}\},$

$\langle \mathbb{P}_0, \mathbb{S}_1, \emptyset\rangle \overset{L}{\rightsquigarrow}{}^{*} \langle \mathbb{P}'_1, _, _\rangle, \quad and \quad \langle \mathbb{P}_0, \mathbb{S}_2, \emptyset\rangle \overset{L}{\rightsquigarrow}{}^{*} \langle \mathbb{P}'_2, _, _\rangle,$

$\mathbb{P}'_1(p_0) = \langle v, _, _\rangle, \quad and \quad \mathbb{P}'_2(p_0) = \langle v', _, _\rangle,$

then $v = v'$.

Proof. By induction on the steps $\overset{L}{\rightsquigarrow}{}^{*}$ of the first execution, and Lemma 11 and Lemma 12.

Definition 8 *For all* $\mathbb{P}$, Γ, $\mathcal{O}$, $\mathcal{M}$, $\mathcal{B}$ *and* c,

$\mathbb{P} \leq_{\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{B}} c$ *iff*

$\forall p, e.$

$\mathbb{P}(p) = \langle e, _, _\rangle \wedge p \notin \mathcal{B} \Rightarrow$

$\exists c'.\ \Gamma, \mathcal{O}, \mathcal{M}, _, _ \vdash e : \langle c', _, _\rangle \wedge \ c' \sqsubseteq c$

Definition 9 *For all* $\mathbb{N}$, Γ, *and* c, $\mathbb{N} \leq_{\Gamma} c$ *iff*

$\forall p, m, id, v, c'.$

$\mathbb{N}(p, id, m, v) \neq \emptyset \wedge$

$\Gamma(m) = _, _ \rightarrow \langle c', _, _ \rangle \Rightarrow$

$c' \sqsubseteq c$

Definition 10 *For all* $\mathbb{P}$, $\mathbb{P}'$, *and* $\mathcal{B}$, $\mathbb{P} \equiv_{\mathcal{B}} \mathbb{P}'$ *iff*

$\forall p.\ \ p \notin \mathcal{B} \Rightarrow \mathbb{P}(p) = \mathbb{P}'(p)$

Lemma 11 *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, $\mathcal{B}$, C, c, $\mathbb{P}$, $\mathbb{S}$, $\mathbb{N}$, $\mathbb{P}'$, $\mathbb{S}'$, *and* $\mathbb{N}'$, *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C,$

$\Gamma(m_0) = _, _ \rightarrow \langle c, _, _ \rangle,$

$\mathbb{P} \leq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} c,$

$\mathbb{N} \leq_{\Gamma} c,$

$\langle \mathbb{P}, \mathbb{S}, \mathbb{N} \rangle \overset{l}{\leadsto} \langle \mathbb{P}', \mathbb{S}', \mathbb{N}' \rangle,$

then

$\mathbb{P}' \leq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} c$, *and* $\mathbb{N}' \leq_{\Gamma} c$

Proof. By case analysis on the step $\overset{l}{\leadsto}$. And then inversion on the type of the stepping expression and then Lemma 38.

Case Op:

Trivial. By the rule ValT, values have the $\bot$ confidentiality. By the rules Op and OpT, the confidentiality stays as $\bot$.

Case Seq:

By the rules Seq and SeqT, confidentiality can only decrease.

Case IfThen:

By the rules If and IfT, the type changes from the join of two types to one of them. Confidentiality can only decrease.

Case IfElse:

Similar to the case IfThen.

Case ThisCall:

We note that this-calls are tail calls. Therefore, $\mathcal{R} = []$.

For $\mathbb{P}$: The expression in the post-state is the value $\bot$. The confidentiality of a value is $\bot$. Thus, the confidentiality of the result is less than c.

For $\mathbb{N}$: The expression $m(v)$ is typed with confidentiality less than c. Thus, by the rule ThisCallT, the confidentiality of the return type of m is less than c.

Case ThisCallExec:

The invariant on $\mathbb{P}$ is proved using the invariant on $\mathbb{N}$.

Case ThisCallByz:

By Lemma 35, the confidentiality of the return type of all the split methods is less than c.

Case ObjCall:

The result is a value. The confidentiality of a value is $\bot$. Confidentiality can only decrease.

Case ObjReCall:

Similar to ObjCall.

Case ObjCallByz:

Similar to ObjCall.

Lemma 12 *For all* C, Γ, $\mathcal{O}$, $\mathcal{M}$, C, $\mathcal{B}$, c, $\mathbb{P}_1$, $\mathbb{P}'_1$, $\mathbb{S}_1$, $\mathbb{N}$, $\mathbb{P}_2$, $\mathbb{P}'_2$, $\mathbb{S}'_1$, $\mathbb{N}'$, $\mathbb{S}_2$, $\mathbb{S}'_2$ *and* $\mathbb{N}'_2$, *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$,

$\mathit{SIntegrity}(\mathcal{O} \mid_\Gamma c) \sqsubseteq \{\mathcal{B}\}$,

$\mathbb{P}_1 \leq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} c$,

$\langle \mathbb{P}_1, \mathbb{S}_1, \mathbb{N} \rangle \overset{l}{\rightsquigarrow} \langle \mathbb{P}'_1, \mathbb{S}'_1, \mathbb{N}' \rangle$,

$\mathbb{P}_1 \equiv_\mathcal{B} \mathbb{P}_2$,

$\langle \mathbb{P}_2, \mathbb{S}_2, \mathbb{N} \rangle \overset{l}{\rightsquigarrow} \langle \mathbb{P}'_2, \mathbb{S}'_2, \mathbb{N}'_2 \rangle$,

$\mathbb{S}_1 \mid_\Gamma c = \mathbb{S}_2 \mid_\Gamma c$,

then

$\mathbb{P}'_1 \equiv_\mathcal{B} \mathbb{P}'_2$, $\mathbb{N}'_2 = \mathbb{N}'$, *and* $\mathbb{S}'_1 \mid_\Gamma c = \mathbb{S}'_2 \mid_\Gamma c$.

Proof. Case analysis on the step $\langle \mathbb{P}, \mathbb{S}_1, \mathbb{N} \rangle \overset{l}{\rightsquigarrow} \langle \mathbb{P}', \mathbb{S}'_1, \mathbb{N}' \rangle$:

The only interesting cases are ObjCall, ObjReCall and ObjCallByz. The rest are trivial.

Case ObjCall:

If there is no process in $\{\overline{p}\}$ that is not in $\mathcal{B}$, then the expressions of these processes in the post-states are not important. Further, if the state of the accessed object in the two steps

are the same in the pre-state, they are the same in the post-state as well. If there is such a non-Byzantine process in p, from $p \notin \mathcal{B}$ and $\mathbb{P} \leq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} c$, we have that the confidentiality type of the expression of p is less than c. By Lemma 39, the redex $o.m(v)$ is less confidential than c. By the rules ObjCallT and FieldT, we have that if $\Gamma(o) = \langle c', _, _ \rangle$ then $c' \sqsubseteq c$. From $c' \sqsubseteq c$ and $\mathbb{S}_1 |_\Gamma c = \mathbb{S}_2 |_\Gamma c$, we have $\mathbb{S}_1(o) = \mathbb{S}_2(o)$. Therefore, by the update in the rule ObjCall, the resulting states are equal: $\mathbb{S}'_1(o) = \mathbb{S}'_2(o)$ and the resulting expressions v' in the two steps are the same.

Case ObjReCall:

Similar to the case ObjCall, we can show that if the process p is non-Byzantine, then $\mathbb{S}_1(o) = \mathbb{S}_2(o)$. Therefore, the resulting expressions v' of the two steps are the same.

Case ObjCallByz:

If $p \in \mathcal{B}$, then its post-state is not important.

If $p \notin \mathcal{B}$, similar to the case ObjCall, we can show that if $\Gamma(o) = \langle c', _, _ \rangle$ then $c' \sqsubseteq c$. Thus, $o \in \mathcal{O} |_\Gamma c$. Therefore, from $\mathit{SIntegrity}(\mathcal{O} |_\Gamma c) \sqsubseteq \{\mathcal{B}\}$, for $\mathcal{O}(o) = \langle _, Q, _ \rangle$, we have $\mathit{SIntegrity}(Q) \sqsubseteq \{\mathcal{B}\}$. This is in contradiction with the assumption of this step.

2.13.2 Integrity Non-Interference

Theorem 13 (Integrity Non-interference) *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, C, $\mathcal{B}$, i, $\mathbb{S}_1$, $\mathbb{S}_2$, $\mathbb{P}'_1$, $\mathbb{P}'_2$, L, v *and* v', *if*

$$\Gamma, \mathcal{O}, \mathcal{M} \vdash C, \quad \textit{and} \quad \Gamma(m_0) = _, _ \rightarrow \langle _, i, _ \rangle,$$

$$\mathbb{S}_1 |_\Gamma i = \mathbb{S}_2 |_\Gamma i, \quad \textit{and} \quad \mathit{SIntegrity}(\mathcal{O} |_\Gamma i) \sqsubseteq \{\mathcal{B}\},$$

$$\langle \mathbb{P}_0, \mathbb{S}_1, \emptyset \rangle \overset{L}{\rightsquigarrow}^* \langle \mathbb{P}'_1, _, _ \rangle, \quad \textit{and} \quad \langle \mathbb{P}_0, \mathbb{S}_2, \emptyset \rangle \overset{L}{\rightsquigarrow}^* \langle \mathbb{P}'_2, _, _ \rangle,$$

$$\mathbb{P}'_1(p_0) = \langle v, _, _\rangle, \quad \textit{and} \quad \mathbb{P}'_2(p_0) = \langle v', _, _\rangle,$$

then $v = v'$.

Proof. By induction on the steps $\overset{L}{\rightsquigarrow}^*$ of the first execution, and Lemma 16 and Lemma 17.

Definition 14 *For all* $\mathbb{P}$, Γ, $\mathcal{O}$, $\mathcal{M}$, i *and* $\mathcal{B}$,

$\mathbb{P} \geq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} i$ *iff*

$\forall p, e.$

$$\mathbb{P}(p) = \langle e, _, _\rangle \wedge p \notin \mathcal{B} \Rightarrow$$

$$\exists i'.\ \Gamma, \mathcal{O}, \mathcal{M}, _, _ \vdash e : \langle _, i', _\rangle \ \wedge$$

$$i' \sqsubseteq i$$

Definition 15 *For all* $\mathbb{N}$, Γ, *and* i,

$\mathbb{N} \geq_{\Gamma} i$ *iff*

$\forall p, m, id, v, i'.$

$$\mathbb{N}(p, id, m, v) \neq \emptyset \ \wedge$$

$$\Gamma(m) = _, _ \rightarrow \langle _, i', _\rangle \Rightarrow$$

$$i' \sqsubseteq i$$

Lemma 16 *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, $\mathcal{B}$, C, i, $\mathbb{P}$, $\mathbb{S}$, $\mathbb{N}$, $\mathbb{P}'$, $\mathbb{S}'$, *and* $\mathbb{N}'$, *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$, $\Gamma(m_0) = _, _ \rightarrow \langle _, i, _ \rangle$,

$\mathbb{P} \geq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} i$, $\mathbb{N} \geq_{\Gamma} i$,

$\langle \mathbb{P}, \mathbb{S}, \mathbb{N} \rangle \overset{l}{\rightsquigarrow} \langle \mathbb{P}', \mathbb{S}', \mathbb{N}' \rangle$, *then*

$\mathbb{P}' \geq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} i$, *and* $\mathbb{N}' \geq_{\Gamma} i$

Proof. Similar to Lemma 11. In contrast to confidentiality that decreases, integrity increases with the steps.

Lemma 17 *For all* C, Γ, $\mathcal{O}$, $\mathcal{M}$, C, $\mathcal{B}$, i, $\mathbb{P}_1$, $\mathbb{P}_1'$, $\mathbb{S}_1$, $\mathbb{N}$, $\mathbb{P}_2$, $\mathbb{P}_2'$, $\mathbb{S}_1'$, $\mathbb{N}'$, $\mathbb{S}_2$, $\mathbb{S}_2'$ *and* $\mathbb{N}_2'$, *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$,

$\{\mathcal{B}\} \sqsubseteq \mathit{SIntegrity}(\mathcal{O} \,|_{\Gamma}\, i)$,

$\mathbb{P}_1 \geq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} i$,

$\langle \mathbb{P}_1, \mathbb{S}_1, \mathbb{N} \rangle \overset{l}{\rightsquigarrow} \langle \mathbb{P}_1', \mathbb{S}_1', \mathbb{N}' \rangle$,

$\mathbb{P}_1 \equiv_{\mathcal{B}} \mathbb{P}_2$,

$\langle \mathbb{P}_2, \mathbb{S}_2, \mathbb{N} \rangle \overset{l}{\rightsquigarrow} \langle \mathbb{P}_2', \mathbb{S}_2', \mathbb{N}_2' \rangle$,

$\mathbb{S}_1 \,|_{\Gamma}\, i = \mathbb{S}_2 \,|_{\Gamma}\, i$,

then

$\mathbb{P}'_1 \equiv_{\mathcal{B}} \mathbb{P}'_2,$

$\mathbb{N}'_2 = \mathbb{N}',$ *and*

$\mathbb{S}'_1 \,|_{\Gamma}\, i = \mathbb{S}'_2 \,|_{\Gamma}\, i.$

Proof. Similar to Lemma 12. In contrast to the fact that objects with confidentiality lower than c have the same state, we have that objects with integrity higher than i have the same state.

2.13.3 Availability Non-Interference

Theorem 18 (Availability Non-interference) *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, C, $\mathcal{B}$, a, $\mathbb{S}_1$, $\mathbb{S}_2$, $\mathbb{P}'_1$, $\mathbb{P}'_2$, L, v *and* v', *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C,$ *and* $\Gamma(m_0) = _, _ \rightarrow \langle _, _, a\rangle,$

$\mathbb{S}_1 \,|_{\Gamma}\, a = \mathbb{S}_2 \,|_{\Gamma}\, a,$ *and* $SIntegrity(\mathcal{O} \,|_{\Gamma}\, a) \sqsubseteq \{\mathcal{B}\},$

$\langle \mathbb{P}_0, \mathbb{S}_1, \emptyset\rangle \overset{L}{\leadsto}{}^{*} \langle \mathbb{P}'_1, _, _\rangle,$ *and* $\langle \mathbb{P}_0, \mathbb{S}_2, \emptyset\rangle \overset{L}{\leadsto}{}^{*} \langle \mathbb{P}'_2, _, _\rangle,$

$\mathbb{P}'_1(p_0) = \langle v, _, _\rangle,$ *and* $\mathbb{P}'_2(p_0) = \langle v', _, _\rangle,$

then $v = v'$.

Proof. By induction on the steps $\overset{L}{\leadsto}{}^{*}$ of the first execution, and Lemma 21 and Lemma 22.

Definition 19 *For all* $\mathbb{P}$, Γ, $\mathcal{O}$, $\mathcal{M}$, a *and* $\mathcal{B}$,

$\mathbb{P} \geq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} a$ *iff*

$\forall p, e.$

$\mathbb{P}(p) = \langle e, _, _ \rangle \wedge p \notin \mathcal{B} \Rightarrow$

$\exists a'.\ \Gamma, \mathcal{O}, \mathcal{M}, _, _ \vdash e : \langle _, _, a' \rangle\ \wedge$

$a' \sqsubseteq a$

Definition 20 *For all* $\mathbb{N}$, Γ, *and* a,

$\mathbb{N} \geq_\Gamma a$ *iff*

$\forall p, m, id, v, a'.$

$\mathbb{N}(p, id, m, v) \neq \emptyset\ \wedge$

$\Gamma(m) = _, _ \to \langle _, _, a' \rangle \Rightarrow$

$a' \sqsubseteq a$

Lemma 21 *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, $\mathcal{B}$, C, a, $\mathbb{P}$, $\mathbb{S}$, $\mathbb{N}$, $\mathbb{P}'$, $\mathbb{S}'$, *and* $\mathbb{N}'$, *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$, $\Gamma(m_0) = _, _ \to \langle _, _, a \rangle$,

$\mathbb{P} \geq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} a$, $\mathbb{N} \geq_\Gamma a$, $\langle \mathbb{P}, \mathbb{S}, \mathbb{N} \rangle \overset{l}{\rightsquigarrow} \langle \mathbb{P}', \mathbb{S}', \mathbb{N}' \rangle$,

then

$\mathbb{P}' \geq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} a$, *and* $\mathbb{N}' \geq_\Gamma a$

Proof. Similar to Lemma 16.

Lemma 22 *For all* C, Γ, $\mathcal{O}$, $\mathcal{M}$, C, $\mathcal{B}$, a, $\mathbb{P}_1$, $\mathbb{P}'_1$, $\mathbb{S}_1$, $\mathbb{N}$, $\mathbb{P}_2$, $\mathbb{P}'_2$, $\mathbb{S}'_1$, $\mathbb{N}'$, $\mathbb{S}_2$, $\mathbb{S}'_2$ *and* $\mathbb{N}'_2$, *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$,

$\{\mathcal{B}\} \sqsubseteq \mathit{SIntegrity}(\mathcal{O} \,|_\Gamma\, a)$,

$\mathbb{P}_1 \geq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} a$,

$\langle \mathbb{P}_1, \mathbb{S}_1, \mathbb{N} \rangle \overset{l}{\rightsquigarrow} \langle \mathbb{P}'_1, \mathbb{S}'_1, \mathbb{N}' \rangle$,

$\mathbb{P}_1 \equiv_{\mathcal{B}} \mathbb{P}_2$,

$\langle \mathbb{P}_2, \mathbb{S}_2, \mathbb{N} \rangle \overset{l}{\rightsquigarrow} \langle \mathbb{P}'_2, \mathbb{S}'_2, \mathbb{N}'_2 \rangle$,

$\mathbb{S}_1 \,|_\Gamma\, a = \mathbb{S}_2 \,|_\Gamma\, a$,

then

$\mathbb{P}'_1 \equiv_{\mathcal{B}} \mathbb{P}'_2$,

$\mathbb{N}'_2 = \mathbb{N}'$, *and*

$\mathbb{S}'_1 \,|_\Gamma\, a = \mathbb{S}'_2 \,|_\Gamma\, a$.

Proof. Similar to Lemma 22.

2.13.4 Integrity Resilience

Theorem 23 (Integrity Resilience) *For all* v, Γ, $\mathcal{O}$, $\mathcal{M}$, C, i, $\mathcal{B}$, $\mathbb{P}$, *and* v', *if*

$\langle m_0(v_0), S_0 \rangle \rightarrow^* \langle v, _ \rangle$,

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C, \quad \Gamma(m_0) = _, _ \rightarrow \langle _, i, _ \rangle, \;$ *and* $\; i \sqsubseteq \{\mathcal{B}\}$,

$\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle \rightsquigarrow^* \langle \mathbb{P}, _, _ \rangle, \;$ *and* $\; \mathbb{P}(p_0) = \langle v', _, _ \rangle$, $v' \neq \bot$,

then $v' = v$.

Proof. Immediate from Lemma 29 and Lemma 30.

Definition 24 *An execution is* macro-step *if a step is taken by all replicating processes before the next step is taken.*

Lemma 25 *For every execution,* $\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle \overset{L}{\rightsquigarrow}^* \langle \mathbb{P}, \mathbb{S}, \mathbb{N} \rangle$*, there is a macro-step execution* $\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle \overset{L'}{\rightsquigarrow}^* \langle \mathbb{P}, \mathbb{S}, \mathbb{N} \rangle$ *where* L' *is a permutation of* L*.*

Proof. The macro-step execution can be incrementally constructed from the given execution by commuting step to the left. The steps taken by OP, SEQ, IFTHEN, IFELSE, and OBJCALLBYZ only change the expression of the current process. Therefore, they can freely commute with steps of other processes. A step taken by THISCALL and THISCALLBYZ, can be moved left. Moving them left only populates the set of messages $\mathbb{N}$ earlier; these messages stay in $\mathbb{N}$ before they are consumed by a subsequent THISCALLEXEC step. A step taken by THISCALLEXEC can be moved left until right after the THISCALL steps that issue the call. They cannot be moved further left as they will not stay enabled due to missing messages. A step taken by OBJCALL can be taken left because the previous call to the same object if any is already executed in the steps that are taken to the left. A step taken by OBJRECALL can be taken left until right after the OBJCALL step that first executes the call. It cannot be moved further left as it will not stay enabled due to missing identifier in the recorded calls.

Definition 26 *A step of a process is a* tail step *if it is not followed by a this-call. An execution is tail-cut when its tail steps are removed.*

Lemma 27 *For every execution,* $\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle \overset{L}{\rightsquigarrow}{}^{*} \langle \mathbb{P}, \mathbb{S}, \mathbb{N} \rangle$*, and its tail-cut execution* $\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle \overset{L'}{\rightsquigarrow}{}^{*} \langle \mathbb{P}', \mathbb{S}', \mathbb{N} \rangle$*, if* $\mathbb{P}(p_0) = \langle v, _, _ \rangle$*,* $v \neq \bot$*, then* $\mathbb{P}'(p_0) = \langle v, _, _ \rangle$*.*

Proof. The computation of tail steps are only local to the executing process. Therefore, removing them does not have any effect on the end result of the execution.

Definition 28 *For all* $\mathbb{P}$*,* $\mathbb{S}$*,* Γ*,* $\mathcal{O}$*,* $\mathcal{M}$*,* $\mathcal{B}$*, and* i*,*

$\mathbb{P}, \mathbb{S} \geq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} i$ *iff* $\forall p, e.$

$$\mathbb{P}(p) = \langle e, _, _ \rangle \wedge p \notin \mathcal{B} \wedge e \neq \bot \Rightarrow$$

$$\exists i'.\ \Gamma, \mathcal{O}, \mathcal{M}, _, _ \vdash e : \langle _, i', _ \rangle \ \wedge \ \ i' \sqsubseteq i \ \wedge$$

$$\langle m_0(v_0), S_0 \rangle \rightarrow^* \langle e, S \rangle$$

where

$$let\ [\overline{o \mapsto \langle s, r \rangle}] := \mathbb{S}\ in\ S = [\overline{o \mapsto s}].$$

Lemma 29 *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, C, $\mathcal{B}$, i, $\mathbb{P}$, *and* v, *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$,

$\Gamma(m_0) = _, _ \to \langle _, i, _ \rangle$,

$i \sqsubseteq \mathcal{B}$

$\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle \rightsquigarrow^* \langle \mathbb{P}, _, _ \rangle$, *and*

$\mathbb{P}(p_0) = \langle v, _, _ \rangle$, $v \neq \bot$

then $\langle m_0(v_0), S_0 \rangle \to^* \langle v, _ \rangle$.

Proof. Based on Lemma 25 and Lemma 27, we consider a macro-step tail-cut execution that results in the value v.

The proof is by induction on the macro-steps of the macro-step execution. The macro-steps are (1) single Op, Seq, IfThen, IfElse, or ObjCallByz steps, (2) the sequence of an ObjCall step and multiple ObjReCall steps for the same object call, and (3) a sequence of ThisCall and ThisCallByz steps and then a sequence of ThisCallExec steps for the same this-call. The inductive invariant is defined in Definition 28. This invariant is a generalization of the invariant of Lemma 16. It adds the relation with the sequential execution to the invariant. The following proof focuses on the added conditions.

Macro steps 1:

Case Op:

Immediate from the rules Op and SOp.

Case Seq:

Immediate from the rules Seq and SSeq.

Case IfThen:

Immediate from the rules IfThen and SIfThen.

Case IfElse:

Immediate from the rules IfElse and SIfElse.

Case ObjCallByz:

Similar to the same case in Lemma 16, this case cannot happen.

Macro steps 2:

A ObjCall step and then a sequence of ObjReCall steps:

For the ObjCall step, from the conditions of the ObjCall rule for the execution of the method call, and from the relation of the concrete state $\mathbb{S}$ and the abstract state S, it can be shown that an SObjCall step can be taken for each process in $\{\overline{p}\}$. For the subsequent ObjReCall steps, from the storage of the object state and the result in the ObjCall step and their retrieval in the ObjReCall steps, it can be shown that an SObjCall step can be taken for each ObjReCall step.

Macro steps 3:

A sequence of ThisCall and ThisCallByz steps and then a sequence of ThisCallExec steps for the same this-call:

The ThisCall and ThisCallByz steps add messages for this-calls to the set of messages $\mathbb{N}$. For the messages added by non-Byzantine principals in the ThisCall steps, there is a corresponding sequential execution. That does not necessarily hold for the messages that are added by ThisCallByz steps. To show that the method call received by a This-

CallExec step has a corresponding sequential execution, we need to show that there is at least one p in the senders q' that is not in $\mathcal{B}$. Let $\mathcal{M}(m) = \langle _, \{\overline{q}\}\rangle$. We have that there is a q in the $\{\overline{q}\}$ that is a subset of q'. From the rule MethodT, we have $\mathit{CIntegrity}(\{\overline{q}\}) \sqsubseteq i_1$ where i_1 is the integrity type of the parameter. By Lemma 37, we have $i_1 \sqsubseteq i$. We also have $i \sqsubseteq \{\mathcal{B}\}$. Therefore, we have $\mathit{CIntegrity}(\{\overline{q}\}) \sqsubseteq \{\mathcal{B}\}$. Therefore, there is at least one process in q that is not Byzantine. Thus, there is at least one process p in q' that is not Byzantine.

Lemma 30 *For all e, v, and v', if*

$\langle e, S_0\rangle \rightarrow^* \langle v, _\rangle$, *and*

$\langle e, S_0\rangle \rightarrow^* \langle v', _\rangle$

then $v = v'$.

Proof. Immediate from induction on the steps of the shorter execution.

2.13.5 Availability Resilience

Theorem 31 (Availability Resilience) *For all v, Γ, $\mathcal{O}$, $\mathcal{M}$, C, i, a, $\mathcal{B}$, and $\mathbb{P}$, if*

$\langle m_0(v_0), S_0\rangle \rightarrow^* \langle v, _\rangle$,

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C, \quad \Gamma(m_0) = _, _ \rightarrow \langle _, i, a\rangle$, *and* $i \sqcup a \sqsubseteq \{\mathcal{B}\}$,

then there exists $\mathbb{P}$ such that

$\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset\rangle \rightsquigarrow^* \langle \mathbb{P}, _, _\rangle$, *and* $\mathbb{P}(p_0) = \langle v, _, _\rangle$.

Proof. The proof is by induction on the sequential steps. For each sequential step, we

take macro-step steps for all the replicating processes. The inductive invariant is defined in Definition 32 below. The preservation of the integrity i and the availability a is similar to Lemma 16 and Lemma 21.

Case SOp:

The step Op is taken for all replicating processes.

Case SSeq:

The step Seq is taken for all replicating processes.

Case SIfThen:

The step SIfThen is taken for all replicating processes.

Case SIfElse:

The step SIfElse is taken for all replicating processes.

Case SThisCall:

A sequence of ThisCall steps is taken for each non-Byzantine replicating process $\mathcal{H}$. Then, a sequence of ThisCallExec steps is taken for each target replicating process. By the invariant, the this-calls are typed. Therefore, from the rule ThisCallT, we have $\mathit{Availability}(Q, \mathcal{H}) \sqsubseteq a_1$ where a_1 is the availability of the parameter. From Lemma 36, we have $a_1 \sqsubseteq a_2$ where a_2 is the availability of the return value. From the inductive invariant, we have $a_2 \sqsubseteq a$. From the assumption of the theorem, we have $a \sqsubseteq \{\mathcal{B}\}$. Therefore, by transitivity, we have $\mathit{Availability}(Q, \mathcal{H}) \sqsubseteq \{\mathcal{B}\}$. Therefore, there are enough non-Byzantine principals q' in $\mathcal{H}$ to make a quorum q in Q. Therefore, the condition $q \subseteq q'$ in the rule ThisCallExec is satisfied. Thus, the ThisCallExec steps can be taken.

Case SObjCall:

A ObjCall step is taken by each non-Byzantine replicating process $\mathcal{H}$. Let $\mathcal{O}(o) = \langle Q_1, Q_2 \rangle$. By the invariant, the object calls are typed. Therefore, from the rule ObjCallT, we have $Availability(Q_2, \mathcal{H}) \sqsubseteq a_1$ where a_1 is the availability of the parameter. From the rule FieldT, we have $a_1 \sqsubseteq a_2$ where a_2 is the availability of the return value. From the inductive invariant, we have $a_2 \sqsubseteq a$ From the assumption of the theorem, we have $a \sqsubseteq \{\mathcal{B}\}$. Therefore, by transitivity, we have $Availability(Q_2, \mathcal{H}) \sqsubseteq \{\mathcal{B}\}$. Therefore, there are enough non-Byzantine principals q' in $\mathcal{H}$ to make a quorum q in Q_2. Therefore, the condition $\{\overline{p}\} \subseteq Q_2$ in the rule ObjCall is satisfied. Further, from the FieldT rule, if $\Gamma(o) = \langle H, i'', a'' \rangle$, we have $i'' \sqsubseteq i_2$, $SIntegrity(Q_1) \sqsubseteq i''$, $a'' \sqsubseteq a_2$, $Availability(Q_1, H) \sqsubseteq a''$. From the inductive invariant, we have $i_2 \sqsubseteq i$ and $a_2 \sqsubseteq a$. From the assumption of the theorem, we have $i \sqsubseteq \{\mathcal{B}\}$, and $a \sqsubseteq \{\mathcal{B}\}$. Therefore, by transitivity, we have $SIntegrity(Q_1) \sqsubseteq \{\mathcal{B}\}$, and $Availability(Q_1, H) \sqsubseteq \{\mathcal{B}\}$. Therefore, the other two conditions of the rule ObjCall are satisfied as well. Therefore, the ObjCall step can be taken.

Definition 32 *For all* $\mathbb{P}$, S, e, Γ, $\mathcal{O}$, $\mathcal{M}$, $\mathcal{H}$, $\mathcal{B}$, i *and* a,

$\mathbb{P}, e, S \geq_{\Gamma,\mathcal{O},\mathcal{M},\mathcal{B}} i, a$ *iff*

$\forall p.\ \ p \notin \mathcal{H} \setminus \mathcal{B} \Rightarrow$

$\quad \mathbb{P}(p) = \langle e, _, _ \rangle \wedge\ \exists i', a'.$

$\quad \Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, _ \vdash e : \langle _, i', a' \rangle \wedge\ i' \sqsubseteq i\ \wedge\ a' \sqsubseteq a\ \wedge$

$\langle \mathbb{P}_0, \mathbb{S}_0, \emptyset \rangle \rightsquigarrow^* \langle \mathbb{P}, _, _ \rangle$ *where*

let $[\overline{o \mapsto s}] := S$ *in* $\mathbb{S} = [\overline{o \mapsto \langle s, _ \rangle}]$.

2.13.6 Helper Lemmas

Lemma 33 *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, C, m, τ *and* τ_{x}, *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C,$

$\Gamma(res) = _, \tau \rightarrow \tau,$

$\Gamma(m) = \tau_{\mathsf{x}}, _ \rightarrow _,$

then

$\tau_{\mathsf{x}} \sqsubseteq \tau.$

Proof. The context type only becomes larger from its initial value at the beginning of the method as it reaches the this-method calls. By the rule THISCALLT, both the initial context type and the parameter type of a this-method is larger than the calling context type. All methods transitively call the final *res* method. Thus, the parameter type of *res* is larger than the initial context type of every this-method.

Lemma 34 *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, C, m, τ *and* τ', *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C,$

$\Gamma(res) = _, \tau \rightarrow \tau,$

$\Gamma(m) = _, _ \rightarrow \tau',$

then

$\tau \sqsubseteq \tau'.$

Proof. All methods transitively call the final *res* method. All the this-calls are tail calls. The proof is by induction on the length of the call sequence from m to the *res* method.

For both rules SeqT and IfT, the result of the join is more than the operands. If there are mutually recursive functions, we consider them in one step. Consider two mutually recursive function with return types τ' and τ'' where the former is closer to the *res* method. We have $\tau \sqsubseteq \tau'$. Because of the mutually recursive calls, we have $\tau'' \sqsubseteq \tau'$, and $\tau' \sqsubseteq \tau''$. Therefore, we have $\tau \sqsubseteq \tau''$.

Lemma 35 *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, C, m, τ *and* τ', *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$,

$\Gamma(m_0) = _, _ \rightarrow \tau$,

$\Gamma(m) = _, _ \rightarrow \tau'$,

then

$\tau' \sqsubseteq \tau$.

Proof. Similar to Lemma 34.

Lemma 36 *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, C, m, τ *and* τ', *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$,

$\Gamma(m) = \tau_{\mathsf{x}}, \tau' \rightarrow \tau$,

then

$\tau_x \sqsubseteq \tau$, *and*

$\tau' \sqsubseteq \tau$.

Proof. Immediate from Lemma 33, Lemma 34 and the condition $\tau' \sqsubseteq \tau$ in METHODT.

Lemma 37 *For all* Γ, $\mathcal{O}$, $\mathcal{M}$, C, m, m', τ *and* τ', *if*

$\Gamma, \mathcal{O}, \mathcal{M} \vdash C$,

$\Gamma(m_0) = _, _ \rightarrow \tau$,

$\Gamma(m) = \tau_{\mathsf{x}}, \tau' \rightarrow _$,

then

$\tau_x \sqsubseteq \tau$, *and*

$\tau' \sqsubseteq \tau$.

Proof. Immediate from Lemma 36 and Lemma 35.

Lemma 38 *For all* $\mathcal{O}$, $\mathcal{M}$, $\mathcal{H}$, τ_{x}, $\mathcal{R}$, e_1, e_2, τ_1, τ_2, τ, *and* τ', *if*

$\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e_1 : \tau_1$,

$\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash e_2 : \tau_2$,

$\tau_2 \sqsubseteq \tau_1$,

$\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash \mathcal{R}[e_1] : \tau$,

then

$\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash \mathcal{R}[e_2] : \tau'$,

$\tau' \sqsubseteq \tau$.

Proof. Induction on $\mathcal{R}$ and then inversion on $\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash \mathcal{R}[e] : \tau$.

$\mathcal{R} \oplus e$, $v \oplus \mathcal{R}$: The join result can be only smaller or the same.

$\mathcal{R}; e$: The join result can be only smaller or the same.

$\mathsf{if}\ \mathcal{R}\ \mathsf{then}\ e\ \mathsf{else}\ e$: The join result can be only smaller or the same.

$m(\mathcal{R})$: The smaller type of the argument still satisfies the conditions and the result type is the same.

$o.m(\mathcal{R})$: Similar to this-call case.

$[\,]$: Trivial.

Lemma 39 *For all $\mathcal{O}$, $\mathcal{M}$, $\mathcal{H}$, τ_{x}, $\mathcal{R}$, e, and τ, if*

$\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau_{\mathsf{x}} \vdash \mathcal{R}[e] : \tau$,

then there exists τ' and τ'_{x} such that

$\Gamma, \mathcal{O}, \mathcal{M}, \mathcal{H}, \tau'_{\mathsf{x}} \vdash e : \tau'$,

$\tau' \sqsubseteq \tau$.

$\tau_{\mathsf{x}} \sqsubseteq \tau'_{\mathsf{x}}$.

Proof. Induction on $\mathcal{R}$. For $\oplus$ and ; and if, the result type is the join of operands. For $m([])$ and $o.m([])$, conditions in FieldT and MethodT state that arguments are smaller than the return types.

Chapter 3

Quorum Subsumption for Heterogeneous Quorum Systems

3.1 Introduction

Bitcoin [350] had the promise to democratize the global finance. Globally scattered servers validate and process transactions, and maintain a consistent replication of a ledger. However, the nature of the proof-of-work consensus exhibited disadvantages such as high energy consumption, and low throughput. In contrast, Byzantine replication have always had modest energy consumption. Further, since its advent as PBFT [116], many recent extensions [423, 338, 445, 110, 60, 94, 95] have improved its throughput. However, its basic model of quorums is closed and homogeneous: the set of processes are fixed, and the quorums are assumed to be uniform across processes. Thus, projects such as Ripple [397] and Stellar [334, 307] emerged to bring heterogeneity and openness to Byzantine quorum

systems. They let every process declare its own set of quorums, or the processes it trusts called slices, from which quorums are calculated.

In this paper, we first consider a basic model of heterogeneous quorum systems where each process has an individual set of quorums. Then, we consider fundamental questions about their properties. Quorum systems are the foundation of common distributed computing abstractions such as reliable broadcast and consensus. We specify the expected safety and liveness properties for these abstractions. What are the necessary and sufficient properties of heterogeneous quorum systems to support these abstractions? Previous work [311] noted that quorum intersection and weak availability properties are necessary for the quorum system to implement the consensus abstraction. Quorum intersection requires that every pair of quorums overlap at a well-behaved process. The safety of consensus relies on the quorum intersection property of the underlying quorum system: intuitively, if an operation communicates with a quorum, and a later operation communicates with another quorum, a single well-behaved process in their intersection can make the second quorum aware of the first. A quorum system is weakly available for a process if it has a quorum for that process whose members are all well-behaved. Intuitively, the quorum system is available to that process through that quorum. Since a process needs to communicate with at least one quorum to terminate, the liveness properties are dependent on the availability of the quorum system.

The quorum intersection and availability properties are necessary. Are they sufficient as well? In this paper, we prove that they are not sufficient conditions to implement reliable broadcast and consensus. For each abstraction, we present execution scenarios, and

apply indistinguishability arguments to show that any protocol violates at least one of the safety or liveness properties. What property should be added to make the properties sufficient? A less known property is quorum sharing [311]. Roughly speaking, every quorum should include a quorum for all its members. This is a property that trivially holds for homogeneous quorum systems where every quorum is uniformly a quorum of all its members. However, in general, it does not hold for heterogeneous quorum systems. Previous work showed that it also holds for Stellar quorums if Byzantine processes do not lie about their slices.

Since Byzantine processes' quorums is arbitrary, in practice, quorum sharing is too strong. In order to require inclusion only for the quorums of a well-behaved subset of processes, we consider a weaker notion, called quorum subsumption. As we will see, this property lets processes in the included quorum make local decisions while preserving the properties of the including quorum. We precisely capture this property, and show that together with the other two properties, it is sufficient to implement reliable broadcast and consensus abstractions. We present protocols for both reliable broadcast and consensus, and prove that if the underlying quorum system has quorum intersection, availability, and subsumption for certain quorums, then the protocols satisfy the required safety and liveness properties. We also present bunched voting and practical consensus protocols that use finite state and only send and receive finite number of messages. They are refinement of the reliable broadcast and consensus protocols mentioned before in order to be amendable for implementation. In summary, this paper makes the following contributions.

- Properties of quorum-based protocols (section 3.3) and specifications of reliable broadcast and consensus on heterogeneous quorum systems (section 3.4).
- Proof of insufficiency of quorum intersection and availability to solve consensus (subsection 3.5.1) and reliable broadcast (subsection 3.5.2).
- Sufficiency of quorum intersection, quorum availability and quorum subsumption to solve consensus and reliable broadcast. We present protocols for reliable broadcast (subsection 3.6.1) and consensus (subsection 3.6.2), and their proofs of correctness.
- We present protocols (subsection 3.6.3) to solve consensus in practical systems with finite states and messages transmitted.

3.2 Heterogeneous Quorum Systems

A quorum is a subset of processes that are collectively trusted to perform an operation. However, this trust may not be uniform: while a process may trust a part of a system, another process may not trust that same part. In this section, we adopt a general model of quorum systems [294, 311] and its properties. These basic definitions adapt common properties of quorum systems to the heterogeneous setting, and serve as the foundation for theorems and protocols in the later sections. Since we want the theorems to be as strong as possible, we introduce the weak notion of quorum subsumption here.

3.2.1 Processes and Quorums

Processes and Failures.

We consider a set of processes $\mathcal{P}$ partitioned into a set of *well-behaved* processes $\mathcal{W}$ and a set of *Byzantine* processes $\mathcal{B}$. Well-behaved processes follow the given protocol, while Byzantine processes can deviate from the protocol arbitrarily.

Processes communicate by sending each other messages using point-to-point links. We assume that there is a point-to-point link between each pair of processes.

We assume that the network is partially synchronous [160], *i.e.*, that there is an unknown global stabilization time (noted GST) and a known delay Δ such that, after GST, every message sent by a well-behaved process to a well-behaved process is received within time Δ. Before GST, messages may be arbitrarily delayed and lost.

We also assume that processes have local clocks that tick at the same rate.

Heterogeneous Quorum Systems (HQS).

In a heterogeneous quorum system, each process has its own, personal set of quorums. For simplicity, we assume that all the quorums of a process are minimal.

Definition 40 (Heterogeneous Quorum System) *A heterogeneous quorum system $\mathcal{Q}$ is a mapping from processes to sets of sets of processes such that, for every process p, $\mathcal{Q}(p)$ is the set of quorums of p and :*

- *$\mathcal{Q}(p)$ is non-empty and, for every $Q \in \mathcal{Q}(p)$, Q is a non-empty set of processes.*
- *Each quorum of p is minimal among $\mathcal{Q}(p)$ (i.e. if $Q \in \mathcal{Q}(p)$ and $Q' \subset Q$, then $Q' \notin \mathcal{Q}(p)$).*

Figure 4.1 presents an example quorum system. Note that, when the set of Byzantine processes is known, we omit specifying the quorums of Byzantine processes, since their behavior is arbitrary regardless of any quorum they may have.

When obvious from the context, we use $\mathcal{Q}$ to refer to the set of all individual quorums of the system, i.e. the union of all sets in the co-domain of $\mathcal{Q}$. Additionally, we say quorum systems to refer to heterogeneous quorum systems.

Definition 41 (Follower) *A process p is a* follower *of a process p' when there is a quorum $q \in \mathcal{Q}(p)$ that includes p'.*

In dissemination quorum systems (DQS) [327] (and the cardinality-based quorum systems as a special case), quorums are uniform for all processes. Processes have the same set of individual minimal quorums. For example, a quorum system that tolerates f Byzantine failures out of $3f + 1$ processes considers any set of $2f + 1$ processes as a quorum for all processes.

3.2.2 Properties

A quorum system is expected to maintain certain properties in order to provide distributed abstractions such as Byzantine reliable broadcast and consensus. Quorum intersection and quorum availability are well-established requirements for quorum systems. In the following section, we will see their adaption to HQS. Further, we identify a new property we call quorum subsumption that helps achieve the aforementioned abstractions on HQS. Finally, we briefly present a few related quorum systems, and their properties.

Quorum Intersection. Processes store and retrieve information from the quorum system by communicating with its quorums. To ensure that information is properly passed from a quorum to another, the quorum system is expected to maintain a well-behaved process at the intersection of every pair of quorums. For example, in the running example in Figure 4.1, all the quorums of well-behaved processes intersect at at least one of well-behaved processes in $\{1, 3, 4\}$.

Definition 42 (Quorum Intersection) *A quorum system $\mathcal{Q}$ has quorum intersection when every pair of quorums of well-behaved processes in $\mathcal{Q}$ intersect at a well-behaved process, i.e., $\forall p, p' \in \mathcal{W}.\ q \in \mathcal{Q}(p).\ q' \in \mathcal{Q}(p').\ q \cap q' \cap \mathcal{W} \neq \emptyset$*

Quorum Availability. In order to support progress for a process, the quorum system is expected to have at least one quorum for that process whose members are all well-behaved. We say that the quorum system is weakly available for that process. (In the literature, this notion of availability is often unqualified, but we explicitly contrast the weak notion to the strong notion that we will define.) In classical quorum systems, any quorum is a quorum for all processes. This guarantees that if the quorum system is available for a process, it is available for all processes. However, this is obviously not true in a heterogeneous quorum system where quorums are not uniform. In this setting, we weaken the availability property so that it requires only a subset and not necessarily all well-behaved processes to have a well-behaved quorum. In Figure 4.1, $\mathcal{Q}$ is available for the set $\{1, 3, 4\}$: the quorum $\{1, 4\}$ of process 1, and the quorum $\{3, 4\}$ of processes 3 and 4 make them weakly available. Each process in that subset can always communicate with a quorum independently of Byzantine processes.

$$
\begin{aligned}
&\mathcal{P} = \mathcal{W} \cup \mathcal{B}, \quad \mathcal{W} = \{1, 3, 4, 5\}, \quad \mathcal{B} = \{2\} \\
&\mathcal{Q} = \{1 \mapsto \{\{1, 2, 3\}, \{1, 4\}\}, \\
&\qquad 3 \mapsto \{\{3, 4\}, \{1, 3\}\} \\
&\qquad 4 \mapsto \{\{3, 4\}\} \\
&\qquad 5 \mapsto \{\{1, 2, 3, 5\}\}\}
\end{aligned}
$$

Figure 3.1: Quorum System Example

Definition 43 (Weak Availability) *A quorum system is weakly available for a set of processes P when every process in P has at least one quorum that is a subset of the set of well-behaved processes $\mathcal{W}$. A quorum system is available when it is available for a non-empty set of processes.*

If a quorum system is weakly available, there is at least one well-behaved process that can communicate with a quorum independently of Byzantine processes.

With quorum availability introduced, we can consider when a quorum system is unavailable. A quorum system is unavailable for a process when that process has no quorum in $\mathcal{W}$, *i.e.*, the Byzantine processes $\mathcal{B}$ can block every one of its quorums. We generalize this idea in the notion of blocking.

Definition 44 (Blocking Set) *A set of processes P is a blocking set for a process p (or is p-blocking) if P intersects every quorum of p.*

For example, consider cardinality-based quorum systems where the system contains $3f + 1$ processes. Any set of size $f + 1$ is a blocking set for all well-behaved processes, since a set with $f + 1$ processes intersects with any quorum, a set with $2f + 1$ processes. In Figure 4.1, well-behaved process 5 is blocked by $\{2\}$, since its only quorum $\{1, 2, 3, 5\}$ intersect with $\{2\}$

Notice also that the definition does not stipulate that the blocking set is Byzantine, but rather it is more general. The concept of blocking will be useful for designing our protocols in (section 3.6). For now, we prove a lemma for blocking sets. In order to state the lemma, we generalize the notion of availability. Given a set of processes P, we generalize availability for P at the complete set of well-behaved processes $\mathcal{W}$ (Theorem 43) to availability for P at a subset P' of well-behaved processes. We say that a quorum system is weakly available for a set of processes P at a subset of well-behaved processes P' when every process in P has at least one quorum that is a subset of P'.

Lemma 45 *In every quorum system that is weakly available for a set of processes P at P', every blocking set of every process in P intersects P'.*

Proof. Consider a quorum system that is weakly available for P at P', a process p in P, and a set of processes P'' that blocks p. By the definition of available, there is at least one quorum q of p that is a subset of P'. By the definition of blocking set (Theorem 74), q intersects with P''. Hence, P' intersects P'' as well.

■

Quorum subsumption. We now introduce the notion of quorum subsumption.

Definition 46 (Quorum Subsumption) *A quorum system $\mathcal{Q}$ is quorum subsuming for a quorum q when every process in q has a quorum that is included in q, i.e., $\forall p \in q.\ \exists q' \in \mathcal{Q}(p).\ q' \subseteq q$. We say that $\mathcal{Q}$ is quorum subsuming for a set of quorums if it is quorum subsuming for each quorum in the set.*

In Figure 4.1, $\mathcal{Q}$ is quorum subsuming for $\{3, 4\}$: both members in this quorum have the quorum $\{3, 4\}$ that is trivially a subset of itself. However, $\mathcal{Q}$ is not quorum

sender	1	2	3	4
$BCast(m_1)$				
	$Echo(m_1)$		$Echo(m_1)$	$Echo(m_1)$
		$Ready(m_2)$		
	$Ready(m_1)$		$Ready(m_2)$	$Ready(m_2)$
	blocked forever			$Deliver(m_2)$

Table 3.1: Non-termination for Bracha protocol with blocking sets

subsuming for process 1's quorum $\{1, 4\}$: process 4's only quorum $\{3, 4\}$ is not a subset of $\{1, 4\}$.

Quorum subsumption is inspired by and weakens the notion of quorum sharing [311]. Quorum sharing requires the above subsumption property for all quorums. Thus, many quorum systems including Ripple and Stellar do not satisfy it (unless Byzantine processes do not lie about their slices [311].) They can maintain the subsumption property only for quorums of a well-behaved subset of processes. In particular, no requirement can be made for quorums of Byzantine processes. Therefore, we define the weaker notion of quorum subsumption for a subset of quorums, and later show that it is sufficient to implement broadcast and consensus.

In order to make progress, protocols (such as Bracha's Byzantine reliable broadcast [86]) require the members of a quorum to be able to communicate with at least one of their own quorums, or communicate with a subset of processes that contains at least one well-behaved process. Let us see intuitively how quorum subsumption can support liveness properties.

Consider a quorum system $\mathcal{Q}$ for processes $\mathcal{P} = \{1, 2, 3, 4\}$ where the Byzantine processes are $\{2\}$, and $\mathcal{Q}(1) = \{\{1, 3, 4\}\}$, $\mathcal{Q}(3) = \{\{1, 2, 3\}\}$, and $\mathcal{Q}(4) = \{\{2, 3, 4\}\}$. The

quorum system $\mathcal{Q}$ has quorum intersection, and is weakly available for the set $\{1\}$ since there is a well-behaved quorum $\{1, 3, 4\}$ for the process 1.

The classic Bracha protocol begins with the sender broadcasting $\mathit{Echo}(m)$. Well-behaved processes broadcast $\mathit{Echo}(m)$ upon receiving it from the sender. After receiving either $2f+1$ $\mathit{Echo}(m)$ or $f+1$ $\mathit{Ready}(m)$ messages, processes broadcast $\mathit{Ready}(m)$. Finally, after receiving $2f+1$ $\mathit{Ready}(m)$ messages, processes deliver m. In Stellar [307] and follow-up works [311, 185, 104], the check for receiving $\mathit{Ready}(m)$ messages from $f+1$ processes is replaced with receiving $\mathit{Ready}(m)$ messages from a blocking set of the current process.

Let's consider the example execution presented in Table 3.1; it gives an intuition of why the quorum system needs stronger conditions than weak availability. Consider a Byzantine sender who sends $\mathit{BCast}(m_1)$ to process $\{1, 3, 4\}$. Well-behaved processes 1, 3, and 4 send out $\mathit{Echo}(m_1)$ to each other. We let process 1 deliver $\mathit{Echo}(m_1)$ messages from process 1, 3, and 4 first; it then sends out $\mathit{Ready}(m_1)$ messages. We note that the two processes 3, and 4 cannot broadcast $\mathit{Ready}(m_1)$ since they have not received $\mathit{Echo}(m_1)$ from a quorum of their own. Then the Byzantine process 2 sends $\mathit{Ready}(m_2)$ messages to processes 3 and 4.

Since the set $\{2\}$ is blocking for the quorums of both processes 3 and 4, both send out $\mathit{Ready}(m_2)$ messages. These broadcast protocols prevent a process that is ready for a value from getting ready for another value. Therefore, although $\{3\}$ and $\{4\}$ are both blocking sets for the process 1, it cannot become ready for m_2. Process 1 never receives enough *Ready* messages for neither m_1 nor m_2 to deliver a message, and is blocked forever. Now, briefly consider the case where the quorum $\{1, 3, 4\}$ for process 1 had the quorum

subsumption property. Then, processes 3 and 4 could send out $Ready(m_1)$ messages, and eventually process 1 would make progress.

Complete Quorum. We will later see that quorum availability and quorum subsumption are important together for liveness. We succinctly combine the two properties into the notion of complete quorums.

Definition 47 (Complete Quorum) *A quorum q in a quorum system $\mathcal{Q}$ is a complete quorum if all its members are well-behaved, and $\mathcal{Q}$ is quorum subsuming for q.*

In our previous running example Figure 4.1, quorum $\{3,4\}$ is a complete quorum: both of its members are well-behaved and $\mathcal{Q}$ is quorum subsuming for $\{3,4\}$.

Definition 48 (Strong Availability) *A quorum system $\mathcal{Q}$ has strong availability for a subset of processes P when every process in P has at least one complete quorum. We call P a* strongly available set *for $\mathcal{Q}$, and call a member of P a* strongly available process*. We say that $\mathcal{Q}$ is strongly available if it is strongly available for a non-empty set.*

Intuitively, operations stay available at a strongly available process since its complete quorum can perform operations on his behalf in the face of Byzantine attacks. In Figure 4.1, $\mathcal{Q}$ is strongly available for $\{3,4\}$. In contrast, $\mathcal{Q}$ is only weakly available for process 1, since its quorum $\{1,2,3\}$ includes 2 that is not well-behaved, and its other quorum $\{1,4\}$ is well-behaved but not a complete quorum.

By Theorem 75, every blocking set of every strongly available process contains at least one well-behaved process.

3.3 Protocol Implementation

In the subsequent sections, we will see that it is impossible to construct a protocol for Byzantine reliable broadcast and consensus in an HQS given only quorum intersection and quorum availability. After that, we give a protocol for Byzantine reliable broadcast and consensus for an HQS that has quorum intersection and strong availability. We first need a model of quorum-based protocols, and then the exact specifications of the distributed abstractions we aim to design protocols for. In this section, we consider the former.

We consider a modular design for protocols. A protocol is captured as a component that accepts request events and issues response events. A component uses other components as sub-components: it issues requests to them and accepts responses from them. A component stores a state and defines handlers for incoming requests from the parent component, and incoming responses from children components. Each handler gets the pre-state and the incoming event as input, and outputs the post-state and outgoing events, either as responses to the parent or requests to the children components. The outputs of a handler can be deterministically a function of its inputs, or randomized.

Definition 49 (Determinism) *A protocol is deterministic when the outputs of its handlers are a function of the inputs.*

Quorum-based Protocols. A large class of protocols are implemented based on quorum systems. In order to state impossibility results for these protocols, we capture the properties of quorum-based protocols [311, 265] as a few axioms. Our impossibility results concern protocols that adhere to the necessity, sufficiency, and locality axioms.

A process in a quorum-based protocol should process a request only if it can communicate with at least one of its quorums.

Axiom 50 (Necessity of Quorums [311]) *If a well-behaved process p issues a response for a request then there must be a quorum q of p such that p receives at least one message from each member of q.*

In a quorum-based protocol, a process only needs the participation of itself and members of one of its quorums to deliver a message.

Axiom 51 (Sufficiency of Quorums) *For every execution where a well-behaved process p issues a response, there exists an execution where only p and a quorum of p take steps, and p eventually issues the same response.*

We add a remark for Byzantine reliable broadcast (BRB) which has a designated sender process. We will use a slight variant of the sufficiency axiom for BRB that states that there exists an execution where only *the sender*, p and a quorum of p take steps.

A process's local state is only affected by the information that it receives from the members of its quorums.

Axiom 52 (Locality [171]) *The state of a well-behaved process changes upon receiving a message only if the sender is a member of one of its quorums. Given the same pre-state and incoming messages from quorum members, a correct process produces the same post-state and outgoing messages.*

For BRB, we will use a slight variant of the locality axiom that allows processes change state upon receiving messages from *the sender* in addition to members of quorums.

3.4 Protocol Specification

We now define the specification of reliable broadcast and consensus for HQS. The liveness properties are weaker than classical notions since in an HQS, availability might be maintained only for a subset P of well-behaved processes.

Reliable Broadcast. We now define the specification of the reliable broadcast abstraction. The abstraction accepts a single broadcast request from a designated sender (either in the system or a process that is separate from the other processes in system), and issues delivery responses.

Definition 53 (Specification of Reliable Broadcast)

- *(Validity for a set of well-behaved processes P). If a well-behaved process p broadcasts a message m, then every process in P eventually delivers m.*

- *(Integrity). If a well-behaved process delivers a message m from a well-behaved sender p, then m was previously broadcast by p.*

- *(Totality for a set of well-behaved processes P). If a message is delivered by a well-behaved process, then every process in P eventually delivers a message.*

- *(Consistency). No two well-behaved processes deliver different messages.*

- *(No duplication). Every well-behaved process delivers at most one message.*

We also consider a variant of reliable broadcast called federated voting. Similar to reliable broadcast, the abstraction accepts a broadcast request from processes, and issues delivery responses. In contrast to reliable broadcast where there is a dedicated sender, in

federated voting, every process can broadcast a message. The specification of federated voting is similar to that of reliable broadcast except for validity. The messages that well-behaved processes broadcast may not be the same. Therefore, the validity property provides guarantees only when the messages are the same or there is only one sender. The validity property for a set of well-behaved processes P guarantees that if all well-behaved processes broadcast a message m, or only one well-behaved process broadcasts a message m, then every process in P eventually delivers m.

Consensus. We now consider the specification of the consensus abstraction. It accepts propose requests from processes in the system, and issues decision responses.

Definition 54 (Specification of Consensus)

- *(Validity). If all processes are well-behaved, and some process decides a value, then that value was proposed by some process.*
- *(Agreement). No two well-behaved processes decide differently.*
- *(Termination for a set of well-behaved processes P). Every process in P eventually decides.*

3.5 Impossibility

We now present the impossibility results for consensus and Byzantine Reliable Broadcast (BRB). It is known that quorum intersection and quorum availability are necessary conditions [311] to implement consensus and BRB protocols. In this section, we show that while these two conditions are necessary, they are not sufficient.

We consider the information-theoretic settings (Fault axiom [171]), where byzantine processes have unlimited computational power, and can show arbitrary behavior. However, processes communicate only over secure channels so that the recipient knows the identity of the sender. A Byzantine process is unable to impersonate a well-behaved process. This is similar to the classic unauthenticated Byzantine general problem [267], and is advantageous for open decentralized blockchains and HQS, where the trusted authorities including public key infrastructures may not be available.

The two proofs will take a similar approach. First, we assume there does exist a protocol for our distributed abstraction that satisfies all the desired specifications. We then present a quorum system $\mathcal{Q}$ and consider its executions that have quorum intersection and availability in the face of Byzantine attacks. We then show through a series of indistinguishable executions that the protocol cannot satisfy all the desired specifications, leading to a contradiction. The high-level idea is that in the information-theoretic setting, a well-behaved process is not able to distinguish between an execution where the sender is Byzantine and sends misleading messages, and an execution where the relaying process is Byzantine and forwards misleading messages. For example, let p_1, p_2 and p_3 be three processes in the system. When p_3 receives conflicting messages from p_1 through p_2, it does not know whether p_1 or p_2 is Byzantine. This eventually leads to violation of the agreement or validity property of the abstraction.

We consider binary proposals for consensus, and binary values (from the sender) for reliable broadcast. For the consensus abstraction, we succinctly present the values that processes propose as as a vector of values that we call a configuration. If the initial value

of a process is $\perp$ in the configuration, that process is considered Byzantine. Otherwise, the process is well-behaved. For example, a configuration $C = \langle 0, 0, \perp \rangle$ denotes the first and second process proposing zero and the third process being Byzantine.

3.5.1 Consensus

We first consider consensus protocols in HQS.

Theorem 55 *Quorum intersection and weak availability are not sufficient for deterministic quorum-based consensus protocols.*

Proof.

We suppose there is a quorum-based consensus protocol that guarantees validity, agreement, and termination for every quorum system $\mathcal{Q}$ with quorum intersection and weak availability, towards contradiction. Consider a quorum system $\mathcal{Q}$ for processes $\mathcal{P} = \{a, b, c\}$ with the following quorums: $\mathcal{Q}(a) = \{\{a, c\}\}$, $\mathcal{Q}(b) = \{\{a, b\}\}$, $\mathcal{Q}(c) = \{\{b, c\}\}$.

We make the following observations: (1) if all processes are well-behaved, then $\mathcal{Q}$ has quorum intersection and weak availability for $\{a, b, c\}$, (2) if only process a is Byzantine, then $\mathcal{Q}$ preserves quorum intersection, and weak availability for $\{c\}$, (3) if only process c is Byzantine, then $\mathcal{Q}$ preserves quorum intersection, and weak availability for $\{b\}$. Going forward, we implicitly assume termination for weakly available processes.

Now consider the following four configurations as shown in Figure 3.2: $C_0 = \langle 0, 0, 0 \rangle$, $C_1 = \langle 1, 1, 1 \rangle$, $C_2 = \langle 0, 1, \perp \rangle$, and $C_3 = \langle \perp, 1, 1 \rangle$. The goal is now to show a series of executions over the configurations so that at least one property of the protocol is violated.

- We begin with execution E_0 (shown in red) with the initial configuration C_0. All the

messages between a and c are delivered. By termination for weakly available processes and validity, process a decides 0. Additionally, by quorum sufficiency, a can reach this decision with only processes $\{a, c\}$ taking steps.

- Next, we have execution E_1 (shown in blue) with initial configuration C_1. All the messages between b and c are delivered. Again, by termination for weakly available processes and validity, process c decides 1. By quorum sufficiency, c can reach this decision with only processes $\{b, c\}$ taking steps.
- Next, we have execution E_2 as a sequence of E_1 and E_0, with initial configuration C_2. Suppose messages between well-behaved processes a and b are delayed. Byzantine process c first replays E_1 with process b, then replays E_0 with process a. By locality, this cause process a to decide 0. Now let Byzantine process c stay silent, and messages between processes a and b be delivered. By termination for b, agreement and quorum sufficiency, process a makes b decide 0 as well (shown in green).
- Lastly, we have execution E_3 with initial configuration C_3. Suppose messages between b to c are delivered in the beginning. We let processes $\{b, c\}$ replay E_1; thus by locality, c decides 1. Then, Byzantine process a sends messages to b as if it were at the end of E_2. In turn, b decides 0. Thus, agreement is violated as two well-behaved processes decided differently.

■

Indistinguishably. We provide some intuition for the proof construction. Ultimately, the problem lies in process b not being able to distinguish whether process a or process c is the Byzantine process. More specifically, both E_2 and E_3 begin with execution

		a	b	c	
C_0		0	0	0	E_0
C_1		1	1	1	E_1
C_2	E_2	0	1	$\perp$	
C_3	E_3	$\perp$	1	1	

Figure 3.2: Indistinguishable Executions

E_1. Since process b cannot distinguish between the two executions, it does not know which value to decide. If process b believes E_2 is the actual execution, then b should decide 0 to agree with the decision of well-behaved process a. However, if E_3 is the actual execution, then agreement is violated as process c decided 1. Conversely, if process b believes E_3 is the actual execution, then b should decide 1 to agree with the decision of well-behaved process c. Then, if E_2 is the actual execution, agreement is violated as the well-behaved process a decided 0.

We note that this proof could not be constructed if there was quorum subsumption. For example, if the process b adds the quorum $\{a, b, c\}$, then $\mathcal{Q}$ will have quorum subsumption for the quorum $\{a, b, c\}$ of b. However, then by quorum subsumption, there will be no Byzantine process, and the executions E_2 and E_3 cannot be constructed. If the process a adds the quorum $\{a, b\}$, then it will have quorum subsumption. However, then the process a cannot Byzantine process anymore, and the executions E_3 cannot be constructed. Similarly, if the process b adds the quorum $\{b, c\}$, the executions E_2 cannot be constructed.

3.5.2 Byzantine Reliable Broadcast

Now, we prove the insufficiency of quorum intersection and quorum availability for Byzantine reliable broadcast.

For the reliable broadcast abstraction, we represent the initial configuration as an array of values received by the processes from the sender. The sender is a fixed and external process in the executions, and is only used to assign input values for processes in the system, which are captured as the initial configurations. The sender does not take steps in the executions, and processes are not able to distinguish executions based on the sender.

Theorem 56 *Quorum intersection and weak availability are not sufficient for deterministic quorum-based reliable broadcast protocols to provide validity and totality for weakly available processes, and consistency.*

Proof. The proof is similar to the proof for consensus. In fact, we will reuse the construction. There are differences between reliable broadcast and consensus specifications in (1) their validity properties, and (2) their totality and termination properties respectively. The proof can be adjusted for these differences. For reliable broadcast, we need a sender process s who broadcasts a message. In executions that we want a well-behaved process to deliver the message m, we either (1) keep the sender s well-behaved and have it send m, and then apply validity, or (2) have a process deliver m, then apply totality and consistency. The initial configuration represents values received by each process from the sender.

Executions follow those in the previous proof. Message delivery and delays mirror the previous executions. In execution E_0 for configuration C_0, the well-behaved sender s broadcasts 0, and messages between processes a and c are delivered. By validity for

weakly available processes, process a delivers 0, and by quorum sufficiency, only processes $\{a, c\}$ need to take steps. In execution E_1 for configuration C_1, the well-behaved sender s broadcasts 1, and messages between processes b and c are delivered. By validity for weakly available processes, and quorum sufficiency, process c delivers 1, only with $\{b, c\}$ taking steps. In configurations C_2 and C_3, the sender s is Byzantine. The messages between processes a and b are delayed in the beginning. In execution E_2 for configuration C_2, the Byzantine sender s and Byzantine process c replay E_1 with process b, then replay E_0 with process a by locality. Then Byzantine process c stays silent, and messages between processes a and b are delivered. By totality for weakly available processes, since process a delivers 0, then process b will also deliver a value. By consistency, process b delivers 0 as well. In the last execution E_3 for configuration C_3, we let the Byzantine process a stay silent in the beginning, and processes b and c replay E_1. Thus by locality, process c delivers 1. Afterwards, messages between process b and c are delayed, and the Byzantine process a replays E_2. Again, process b cannot distinguish between the two executions E_2 and E_3. Since process a sends the exact same messages to process b as the end of E_2, process b will deliver 0. Thus, consistency between c and b is violated. ■

3.6 Protocols

We just showed that quorum intersection and availability are not sufficient to implement our desired distributed abstractions. Now, we show that quorum intersection and strong availability, our newly introduced property are sufficient to implement both Byzantine reliable broadcast and consensus.

Algorithm 1: Byzantine Reliable Broadcast (BRB)

```
1  Implements: ReliableBroadcast
2    request : broadcast(v)
3    response : deliver(v)
4  Vars:
5    Q                                              ▷ Minimal quorums of self
6    F : Set[P]                                     ▷ The followers of self
7    echoed, readied, delivered : Boolean ← false
8    E, R : V ↦ Set[P] ← ∅
                                       ▷ Set of echoed and readied processes
9  Uses:
10   apl : PointToPointLink
11 upon request broadcast(v) from sender
12   apl request send(p, BCast(v)) for each p ∈ P
13 upon apl response deliver(p′, BCast(v))
14   if ¬echoed then
15     echoed ← true
16     apl request send(p, Echo(v)) for each p ∈ F
17 upon apl response deliver(p′, Echo(v))
18   E(v) ← E(v) ∪ {p′}
19   if ¬readied ∧ ∃q ∈ Q. q ⊆ E(v) then
20     readied ← true
21     apl request send(p, Ready(v)) for
         each p ∈ F
22 upon apl response deliver(p′, Ready(v))
23   R(v) ← R(v) ∪ {p′}
24   if ¬readied ∧ R blocks self then
25     readied ← true
26     apl request send(p, Ready(v)) for
         each p ∈ F
27   if ¬delivered ∧ ∃q ∈ Q. q ⊆ R(v) then
28     delivered ← true
29     response deliver(v)
```

3.6.1 Reliable Broadcast Protocol

In Alg. 1, we adapt the Bracha protocol [86] to show that quorum intersection and strong availability together are sufficient for Byzantine reliable broadcast. The parts that are different from the classical protocol are highlighted in blue.

Each process stores the set of its individual minimal quorums Q, and its set of followers $\mathcal{F}$. It also stores the boolean flags *echoed*, *readied*, and *delivered* which record actions the process has taken to avoid duplicate actions. It further uses point-to-point links *apl* to each of its followers. Upon receiving a request to broadcast a value v (at line 11), the sender broadcasts the value v to all processes (at line 12). Upon receiving the message from the sender (at line 13), a well-behaved process echoes the message among its followers (at line 16) only if it has not already *echoed*. When a well-behaved process receives a quorum of consistent echo messages (at line 17), it sends ready messages to all its followers (at line 21). A well-behaved process can also send a ready message when it receives consistent ready messages from a blocking set (at line 24). When a well-behaved process receives a quorum of consistent ready messages for v (at line 27), it delivers v (at line 29). The implementation of the federated voting abstraction is similar. The only difference is that there can be multiple senders (at line 11).

We prove that this protocol implements Byzantine reliable broadcast when the quorum system satisfies quorum intersection, and strong availability. We remember that strong availability requires both weak availability and quorum subsumption. More precisely, it requires a well-behaved quorum q for a process p, and quorum subsumption for q.

Lemma 57 *No two well-behaved, weakly available processes can send two different ready messages.*

Proof. Recall that a process sends a ready message for a message m upon the satisfaction of one of two conditions: (1) the process receives a quorum of consistent echo messages for m or (2) the process receives a blocking set of consistent ready messages for m. This gives us three cases, each of which we show are not possible. Consider some well-behaved process p.

The first case is where condition (1) is satisfied for two different messages m and m'. However, p is in each of its own quorums and every process echoes at most one message. Thus, p cannot receive a quorum of echoes for both m and m'.

The second case is where condition (1) is satisfied for message m and condition (2) is satisfied for m', without loss of generality. Since p is weakly available, it has a quorum containing only well-behaved processes. This means any p-blocking set contains at least one well-behaved process. Let this process be p_w. We inductively reason that p_w cannot have sent a ready message for m'. The base case is that condition (1) is satisfied for p_w for m'. By quorum intersection, there is some well-behaved process between p's quorum for m and p_w's quorum for m'. This implies that a well-behaved process echoed m to p and m' to p_w. However, a well-behaved process cannot echo two different messages, thus the base case cannot happen. The inductive case is that condition (2) is satisfied for p_w for m'. That is, p_w received a p_w-blocking set of ready messages for m'. This eventually reduces to the base case by applying the same reasoning above; for p_w to receive a blocking set of ready messages, at least one of those processes is well-behaved thus is in the same situation as p_w.

As we showed the base case is impossible, we showed that the second case is impossible as well.

The third case is where condition (2) is satisfied for two different messages m and m'. We can employ the same reasoning we saw in the previous case, showing that condition (2) cannot be satisfied for m'. Essentially, both blocking sets for m and m' must reduce to two quorums of echoes for m and m'. By quorum intersection, these quorums must intersect at a well-behaved process, which cannot echo both m and m'. ■

Theorem 58 *Quorum intersection and strong availability are sufficient to implement Byzantine reliable broadcast.*

This theorem follows from the following five lemmas. In the following lemmas, we consider a quorum system with quorum intersection and strong availability for P.

Lemma 59 *The BRB protocol guarantees consistency.*

Proof. A well-behaved process only delivers a message when it receives a quorum of consistent ready messages. If p_1 delivers m_1 with q_1, and p_2 delivers m_2 with q_2, by quorum intersection, there is well-behaved process p in $q_1 \cap q_2$. The process p sends ready messages with only one value. Thus, $m_1 = m_2$. ■

Lemma 60 *The BRB protocol guarantees validity for P.*

Proof. Consider a well-behaved sender that broadcasts a message m. We show that every process in P eventually delivers m. By availability, every process $p \in P$ has a complete quorum q. Consider a process $p' \in q$. By quorum subsumption, p' has a quorum

$q' \subseteq q$. By availability, all members of q (including q') are well-behaved. Thus, when they receive m from the sender, they all echo it to their followers. The processes in q' have p' as a follower. Thus, p' receives consistent echo messages for m from one of its quorums q'. Thus, p' sends out ready messages for m to its followers. Thus, all processes in q send out ready messages for m to their followers. The processes in q have p as a follower. Therefore, p receives a quorum of consistent ready messages for m from one of its quorums q, and delivers m. Consider a well-behaved sender that broadcasts a message m. We show that every process in P eventually delivers m. The protocol stipulates that a process delivers m upon receiving a quorum of consistent ready messages for m. Thus, it is sufficient to show that this condition is satisfied for every process $p \in P$.

Since the quorum system is strongly available for the set of processes P, every process $p \in P$ has a complete quorum q. We want to show that every process $p' \in q$ sends a ready message to p, satisfying the above condition. Since p is a follower of every process p', p' sends ready to p upon receiving a quorum q' of consistent echo messages.

To show that such a quorum q' exists, we remind that q is a complete quorum and that $p' \in q$. Then by quorum subsumption, p' must have a quorum $q' \subseteq q$. Since q is a complete quorum, every member of q is well-behaved, including those of q'. This implies that when each member of q receives m from the sender,they echo it to their followers. Thus, p' would have received an echo for m from every member of q'. ■

Lemma 61 *The BRB protocol guarantees totality for P.*

Proof. We assume that a well-behaved process p has delivered m, and show that every well-behaved process $p' \in P$ delivers m. We first show that every well-behaved process p'' sends ready messages for m to its followers.

The well-behaved process p delivers m only when it receives a quorum q of ready messages for m. Consider a quorum q_i'' of p''. By quorum intersection, q and q_i'' intersect in at least a well-behaved process p_i. Since p_i is in q_i'', then p_i has p'' as a follower. Let I be the union of the processes p_i for all quorums q_i'' of p''. By construction, the set of processes I are a subset of q, a blocking set of p'', and have p'' as a follower. All well-behaved processes in q send ready messages to their followers. Thus, the set of processes I send ready messages to p''. Thus, p'' receives a blocking set of ready messages, and sends ready messages to its followers.

Thus, every well-behaved process p'' sends ready messages to their followers. By weak availability, $p' \in P$ has a quorum q' with all well-behaved members. Thus, the process p' will receive ready messages from q', and delivers m.

Assume that a well-behaved process p_0 has delivered m. We then want to show that every well-behaved process in P also delivers m. To do this, we first claim that every well-behaved process sends a ready message for m to its followers. It is easy to see that if claim holds, then every process in P delivers m. Since every process in P has a complete quorum q, if every well-behaved process sends a ready message for m, then so will every member of q. Then, every process in P will receive a quorum of consistent ready messages, namely q, and deliver m. Thus, it remains to prove the claim above.

For p_0 to have delivered m, it must have received a quorum q_0 of ready messages for m. Now consider any well-behaved process $p_w \in \mathcal{W}$ and any one of its quorums q_w. Quorums q_0 and q_w must intersect at a well-behaved process p_i, by quorum intersection. Let I be the union of these processes p_i, for all quorums of p_w. That is, I is the set of processes that are in both q_0 and a quorum of p_w. By construction, we have (1) $I \subseteq q_0$ and (2) I is a blocking set for p_w.

Since every member of q_0 sent a ready message to its followers, every member of I sent a ready message to p_w. Thus, p_w receives a blocking set of ready messages, meaning it also sends a ready message to its followers, proving our claim.

■

Lemma 62 *The BRB protocol guarantees no duplication.*

Proof. Since a well-behaved process keeps the *delivered* boolean, it delivers at most one message. ■

Lemma 63 *The BRB protocol guarantees integrity.*

Proof. We assume that a well-behaved process p delivers a message m from a well-behaved sender process p', and prove that the process p' has broadcast m. We first prove that a well-behaved process has received echo messages from at least one of its quorums.

Since the well-behaved process p delivered m, it has received a quorum q_p of ready messages. By the strong availability assumption, P is non-empty; therefore, there exists a process with a quorum q that is well-behaved and quorum subsuming. By quorum intersection, q_p and q intersect at a well-behaved process p_i. The process p_i sent a ready

message (that p received). The well-behaved process p_i sends a ready message only if it either (1) receives echo messages from a quorum of p_i, or (2) receives ready messages from a p_i-blocking set B. The first case is immediately the conclusion with the process p_i. Let us consider the second case. By quorum subsumption, since p_i is in q, p_i has a quorum q_i that is a subset of q. Since B is a blocking set of p_i, B intersects with q_i. Therefore, B intersects with q. The processes in B sent ready messages. Thus, there are processes in q that sent ready messages. Let p_f be the first process in q that sent a ready message. All members of q are well-behaved. A well-behaved process p_f sends a ready message only if it either (1) receives echo messages from a quorum of p_f, or (2) receives ready messages from a p_f-blocking set B'. The first case is immediately the conclusion with the process p_f. Let us consider the second case. Applying the same reasoning as for p_i to p_f derives that B' intersects with q. Thus, p_f has received a ready message from a process in q. Therefore, p_f is not the first process in q to send a ready message. This is a contradiction with the definition of p_f above. We now use the fact that a well-behaved process has received echo messages from at least one of its quorums. By quorum intersection at well-behaved processes, every quorum of a well-behaved process includes a well-behaved process. Thus, a well-behaved process p_w has sent an echo message. A well-behaved process sends an echo message only after receiving it from the sender p'. Since the sender p' is well-behaved, and p_w has received the message m from it, by the integrity of point-to-point links, p' has previously sent m to p_w. Since p' is well-behaved, it has sent m to every process. ∎

3.6.2 Byzantine Consensus Protocol

In this section, we show that quorum intersection and strong availability are sufficient to implement Byzantine consensus. We first present the consensus protocol for heterogeneous quorum systems, and then prove its correctness.

At a high level, the protocol proceeds in rounds with assigned leaders for each. Ballots that carry proposal values are totally ordered. A leader tries to commit its own candidate ballot only after aborting all lower ballots in the system. Leaders use the federated voting abstraction (that we saw in section 3.4) to abort or commit ballots. There may be multiple leaders or Byzantine leaders before GST who may broadcast contradicting abort and commit messages for the same ballot. However, by the consistency property of federated voting, processes agree on aborting or committing ballots.

A ballot b is a pair $\langle r, v\rangle$ of a round number r and a proposed value v. Ballots are totally ordered by first their round numbers, and then their values: a ballot $\langle r, v\rangle$ is below another $\langle r', v'\rangle$, written as $\langle r, v\rangle < \langle r', v'\rangle$, if $r < r'$ or $r = r' \wedge v < v'$. Two ballots $b = \langle r, v\rangle$ and $b' = \langle r', v'\rangle$ are compatible, $b \sim b'$, if they have the same value, *i.e.*, $v = v'$; otherwise, they are incompatible, $b \nsim b'$. We say that a ballot is below and incompatible with another, $b \lessapprox b'$, if $b < b'$ and $b \nsim b'$. For message passing communication, we assume batched network semantics (BNS), where messages issued in an event are sent as a batch, and the receiving process delivers and processes the batch of messages together. (In particular, as we will see later in the correctness proofs, if prepare messages that are sent together are not processed together the validity property can be violated.)

The protocol is similar to SCP [334, 187] in structure; the important difference is that this protocol uses leaders [311] and guarantees termination. Our protocol guarantees termination regardless of Byzantine processes. On the other hand, the SCP protocol guarantees a liveness property called *non-blocking* which requires Byzantine processes to stop. (More precisely, if a process p in the intact set [334, 185] has not yet decided in some execution, then for every continuation of that execution in which all the Byzantine processes stop, the process p eventually decides.)

Algorithm 2: Byzantine Consensus

```
1  Implements: Consensus
2    request : propose(v)
3    response : decide(v)
4  Vars:
5    round : ℕ⁺ ← 0                ▷ Current round number
6    candidate, prepared : ⟨ℕ⁺, V⟩ ← ⟨0, ⊥⟩
7    leader : 𝒫 ← p_0              ▷ current leader
8  Uses:
9    fv : B ↦ ByzantineReliableBroadcast
10   le : EventualLeaderElection
11 upon request propose(v)
12     candidate ← ⟨1, v⟩
13     if self = leader then
14         fv(b′) request broadcast(𝔸) for all
             b′ ≴ candidate
15 upon fv(b′) response deliver(p, 𝔸) for all
   b′ ≴ b where prepared < b
16     prepared ← b
17     if self = leader ∧ prepared = candidate
        then
18         fv(candidate) request broadcast(ℂ)
19 upon fv(b) response deliver(p, ℂ),
   b = prepared ∧ p = leader
20     response decide(b.v)
21 upon timeout triggered
22     le request Complain(round)
23 upon le response new-leader(p)
24     leader ← p
25     round ← round + 1
26     if self = leader then
27         Delay for time Δ
28     start-timer(round)
29     if prepared = ⟨0, ⊥⟩ then
30         candidate ← ⟨round,
31         candidate.v⟩
32     else
33         candidate ← ⟨round,
34         prepared.v⟩
35     if self = leader then
36         fv(b′) request broadcast(𝔸)
             ∀b′ ≴ candidate
```

Each process stores four local variables: *round* is the current round number, *candidate* is the ballot that the process tries to commit, *prepared* is the ballot that the process is safe to discard any ballots lower and incompatible with, and *leader* is the current leader. Each process uses an instance of federated voting for each ballot, and an eventual

leader election module. The latter issues *new-leader* events, and eventually elects a well-behaved process as the leader. (Previous work [311] presented a probabilistic leader election module.)

Upon receiving a proposal request (at line 11), a well-behaved process initializes its candidate ballot to the pair of the first round and its own proposal (at line 12). If the current process **self** is the leader, it tries to prepare its *candidate* by broadcasting abort $\mathbb{A}$ messages for all ballots below and incompatible with *candidate* (at line 14). When a well-behaved process delivers $\mathbb{A}$ messages from the leader for all ballots below and incompatible with some ballot b, and its current *prepared* ballot is below b (at line 15), it sets *prepared* to b (at line 16). If the current process **self** is the leader, and the *prepared* ballot is equal to the *candidate* ballot, then it broadcasts a commit $\mathbb{C}$ message for its *candidate* ballot (at line 18). When a well-behaved process delivers a $\mathbb{C}$ message for a ballot b from the leader, and it has already prepared the same ballot (at line 19), it decides the value of that ballot (at line 20).

To ensure liveness, a well-behaved process triggers a timeout if no value is decided after a predefined time elapses in each round. The process then complains to the leader election module (at line 22). When the leader election module issues a new leader (at line 23), a well-behaved process updates its *leader* variable, and increments the *round* number (at line 25). The leader itself then waits for a time Δ (at line 27) which we will further explain below. The process also resets the timer with a doubled timeout for the next round (at line 28). It then updates the *candidate* ballot: if no value is prepared before, the *candidate* ballot is updated to the new round number and the value of the current *candidate*

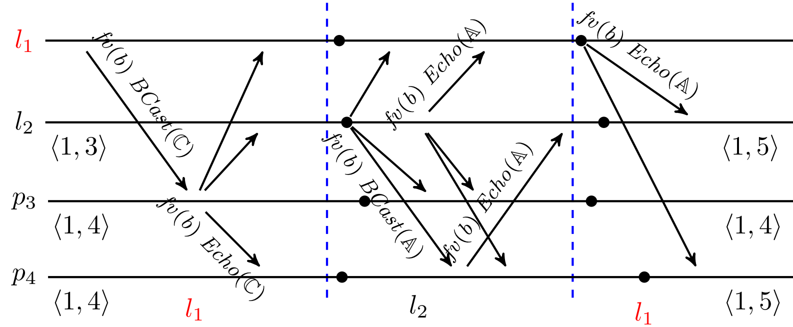


Figure 3.3: Last Minute Attack. $b = \langle 1, 4 \rangle$. The *candidate* of well-behaved leader l_2 is $b' = \langle 2, 3 \rangle$. The votes commit and abort are abbreviated as $\mathbb{C}$ and $\mathbb{A}$. The new leader events are triggered at the black dots at each process. Prepared ballots are shown below the time line for each process.

(at line 31); otherwise, it is updated to the new round number and the value of the *prepared* ballot (at line 34). Then, the leader tries to prepare the *candidate* by aborting below and incompatible ballots similar to the steps above (at line 36).

Let us now explain why delay Δ is needed for termination. Without this delay, a Byzantine leader can perform a last minute attack that we illustrate in Figure 3.3. Consider that we have four processes, one of them is Byzantine, and any set of three processes is a quorum. Let the Byzantine process be the leader l_1, and let the ballot b be prepared. The leader l_1 sends a commit for ballot b to one well-behaved process p_3. Then, p_3 echos commit for b. Then, the timeout for l_1 happens, and the next well-behaved leader l_2 comes up. Without the delay, l_2 may have not prepared b yet (although other well-behaved processes p_3 and p_4 prepared it). Therefore, the ballot b' that l_2 updates its candidate to (at line 34) is not b, and may not be compatible with b. In order to prepare b', the leader l_2 tries to abort b (at line 36) but b cannot be aborted: in order to abort b, a quorum of processes should echo it. However, the well-behaved process p_3 has already echoed commit, and if

the Byzantine process l_1 remains silent, the remaining two well-behaved processes l_2 and p_4 are not a quorum, and cannot abort b. Therefore, l_2 cannot succeed, and the timeout is triggered. Further, if the next leader is the Byzantine process l_1 again, it can repeat the above scenario: it can abort b to prepare a higher ballot b_2, and make a well-behaved process echo commit for b_2, before passing the leadership. The attack can continue infinitely, and delay termination. If the delay Δ is larger than the bounded communication delay after GST, it makes the leader l_2 observe the highest prepared ballot b, and adopt its value as the value of its candidate b_2' (at line 34). When it tries to commit b_2', since it is compatible with b, it does not need abort it. Therefore, it can prepare and commit b_2', and decide. We also note that instead of the delay Δ, the above attack can be avoided if the leader election can provide two successive well-behaved leaders.

Theorem 64 *Quorum intersection and strong availability are sufficient to implement consensus.*

This theorem follows from the following three lemmas where we consider a quorum system with quorum intersection and strong availability for P. They prove that the protocol satisfies the specification of Byzantine consensus that we defined in Theorem 54.

Lemma 65 (Agreement) *The consensus protocol guarantees agreement.*

Proof. We prove agreement by contradiction. Assume that there are two well-behaved processes that decide two different values. A process decides a value at line 20 after delivering a ballot with that value at line 19. Let process p deliver v in ballot b, and let process p' deliver v' in ballot b'. Since ballots are totally ordered, without loss of generality,

we assume that $b \lnapprox b'$. When the process p' delivers v' at line 19, it checks that b' has been prepared. A well-behaved process prepares a new ballot b' at line 16 only after finding all the ballots below and incompatible with b' aborted at line 15. Since $b \lnapprox b'$, the process p' has delivered the $\mathbb{A}$ message for b before preparing b'. However, since process p decides v with b at line 20, it has delivered $\mathbb{C}$ message for b at line 19. Thus, two well-behaved processes deliver conflicting values $\mathbb{A}$ and $\mathbb{C}$ for ballot b, which violates the consistency property of federated voting, a contradiction. ■

Lemma 66 (Validity) *The consensus protocol guarantees validity.*

Proof. We assume that all processes are correct, and that a process has decided a value v. We show that v is proposed by some process. When a well-behaved process decides a value v at line 20, v was delivered in a $\mathbb{C}$ message for a ballot b at line 19. Since all processes are well-behaved including the leader, by the integrity property of federated voting, the leader has broadcast $\mathbb{C}$ for the ballot b at line 18. The ballot b is the leader's *candidate* by the condition at line 17.

We show that the value of every *candidate* ballot can be tracked back to the proposal of a process. The value for the *candidate* ballot is set to either its initial proposal at line 12 and line 31, or adopted from the value of *prepared* at line 34. We consider the two cases in turn. For the first case, the value v is immediately the leader's proposal at line 11. For the second case, v is a value prepared before by the leader. A well-behaved process updates its *prepared* value to b at line 16 when it delivers $\mathbb{A}$ for all $b' \lnapprox b$ at line 15. By the integrity of federated voting, and the assumption that all the processes are well-behaved, leaders have broadcast to abort all ballots b'. Leaders abort ballots below and

incompatible with their *candidate* at line 14 and line 36. By the batched network semantics assumption, all the messages sent by a process in one event to another process are delivered and processed together. Therefore, the ballot b has been the *candidate* ballot of a previous leader at line 14 or line 36. With the same reasoning as above, the *candidate* value can be traced back to either a proposal, or the *candidate* value of a yet previous leader. Since the number of leaders is finite, by a well-founded induction, the *candidate* value is tracked back to the proposal of a process. ■

The protocol guarantees termination: all strongly available processes eventually decide a value. The high-level idea is that by eventual election of a well-behaved leader, and the totality property of federated voting, a well-behaved leader eventually adopts the value of the highest prepared ballot in the system as its candidate Then, that leader can proceed to prepare and commit its candidate.

Lemma 67 (Termination) *The consensus protocol guarantees termination for P.*

Proof. The proof will refer to the notions of locked for federated voting. We first visit these notions, and then describe the proof of termination.

We saw in the explanation of the protocol that there are executions that Byzantine processes can lock a federated voting instance for a certain value (such as commit). Let us remember that example. Consider that we have 4 processes, one of them is Byzantine, and any set of 3 processes is a quorum. Let the Byzantine process send a message m to one well-behaved process. That well-behaved process echos m. If the Byzantine process remains silent, the remaining 2 well-behaved processes are not a quorum, and even if both echo another message m', they cannot make the federated voting deliver m'. This execution

is shown in the first two rounds of messages in Figure 3.3. In this execution, the federated voting instance of b is locked for $m = \mathbb{C}$ by l_1 and p_3, and although l_2 and p_4 try to make it deliver $m' = \mathbb{A}$, it does not. However, if the Byzantine process echos m' as well, then they form a quorum, and can make m' be delivered. Well-behaved processes alone cannot make the instance deliver a different value, but well-behaved and Byzantine processes together can. We call these states of the federated voting abstraction soft-locked.

We note that there are also states of a federated voting instance where even if well-behaved and Byzantine processes work together, they cannot make the instance deliver a different value. We call them hard-locked. As above, consider an execution with 4 processes, and quorums of size 3. Let a Byzantine process send m to two well-behaved processes, and make them echo m. By the protocol, in order to deliver a message, a process needs to receive at least three ready messages. Further, in order to send a ready message, a process needs to receive at least either three echo messages or two ready messages. Therefore, to deliver any other message m', at least 3 processes should echo m'. Since 2 well-behaved processes out of 4 processes already echoed m, and well-behaved processes echo only once, there are not 3 processes to echo m', and make it delivered. The federated voting abstraction is hard-locked for m.

We now consider the proof of termination. By the eventual leader election, there will eventually be a well-behaved leader with a long enough timeout. Let b be the *candidate* ballot of the leader. The leader tries to abort all ballots lower and incompatible with b at line 14 and line 36. We consider three cases based on whether there is a ballot $b' \lnapprox b$ that is locked for commit.

Case 1: There is no such ballot b'. The leader can abort all ballots below and incompatible with b at line 15, and broadcast commit for b at line 18. By the validity of federated voting, all processes in P eventually deliver commit for b at line 19, and decide its value at line 20.

Case 2: There is such a ballot b' that is soft-locked for commit. In order to prepare b, the leader tries to abort b'. If the Byzantine processes cooperate, it can abort b' and prepare b at line 15, and broadcast commit for b at line 18. However, if the Byzantine processes do not cooperate, the leader cannot abort b'. Therefore, the leader does not succeed, and the timeout is triggered. Eventually, a well-behaved process l will be the leader. Since the delay Δ is larger than the bounded communication delay after GST, the leader l observes the highest prepared ballot b at line 15, and adopts its value as the value of its *candidate* b' at line 17. We note that when it tries to prepare b', since b' is compatible with b, it does not need to abort b. Therefore, it can prepare and broadcast commit for b' at line 18. By the validity of federated voting, all processes in P eventually deliver commit for b' at line 19, and decide its value at line 20.

Case 3: There is such a ballot b' that is hard-locked for commit. Ballot b' cannot be aborted. Therefore, no well-behaved process can prepare a ballot with a value incompatible with b'. The current leader will eventually timeout. Further, all ballots $b'' \lnsim b'$ should have been aborted since processes accept commit for b' only if b' is *prepared* at line 19. By the totality of federated voting, eventually a well-behaved leader that has delivered abort for all ballots b'' will be elected. Therefore, that leader finds b' prepared at line 15-16, and adopts its value as the value of its *candidate* ballot b''_2 at line 34. Thus, b''_2 is compatible

with b'. As we saw above, no ballot incompatible with b' can be prepared. Thus, no ballot incompatible with b''_2 can be prepared. Thus, the leader can successfully prepare b''_2 at line 15, and broadcast commit for b''_2 at line 18-17. By the validity of federated voting, all processes in P eventually deliver commit for b''_2 at line 19, and decide its value at line 20.

■

An example execution of the protocol is described in the appendix [290].

3.6.3 Practical Byzantine Consensus Protocol

Algorithm 3: Bunched voting (BV) prepare (1/2)

```
1  Implements: Bunchedvoting
2    request : prepare(b)
3    response : prepared(b)
4  Vars:
5    Q                                                      ▷ Minimal quorums of self
6    𝓕 : Set[𝓟]                                             ▷ The followers of self
7    MaxEchoedPrep, MaxReadiedPrep, MaxDeliveredPrep : ⟨ℕ⁺, V⟩ ← ⟨0, ⊥⟩
8    SecondReadiedPrep : ℕ⁺ ← 0 ⟨ℕ⁺, V⟩ ← ⟨0, ⊥⟩
9    E, R : messages ← ∅
                                              ▷ Set of EchoPrep, ReadyPrep messages
10 Uses:
11   apl : PointToPointLink
12 upon request prepare(b) from sender
13   apl request send(p, Prepare(b)) for each p ∈ 𝓟
14 upon apl response deliver(p', Prepare(b))
15   if MaxEchoedPrep < b ∧ ¬BEchoedCmt ≴ b then
16     MaxEchoedPrep ← b apl request send(p, EchoPrep(b)) for each p ∈ 𝓕
```

In this section, we introduce bunched voting (Alg. 3) and practical Byzantine consensus (Alg. 7) protocols, which are more feasible in the practical systems. They are inspired by concrete stellar consensus protocols in [187]. Each process maintains finite states and

Algorithm 4: Bunched voting (BV) prepare (2/2)

1 **upon** *apl* **response** $deliver(p', EchoPrep(b))$
2 $E \leftarrow E \cup \{deliver(p', EchoPrep(b))\}$
3 **if** $\exists max(b') = b_{max}$ *s.t.* $MaxReadiedPrep < b'$ $\exists q \in Q.\ \forall u \in q, \exists deliver(u, EchoPrep(b_u)) \in E \land b'' \lnsim b_u$ *for every* $b'' \lnsim b'$ **then**
4 **if** $MaxReadiedPrep = \langle r_m, v_m \rangle \land v_m \neq b_{max}.v$ **then**
5 $SecondReadiedPrep \leftarrow r_m$
6 $MaxReadiedPrep \leftarrow b_{max}$
7 *apl* **request** $send(p, ReadyPrep\ (SecondReadiedPrep, b_{max}))$ for each $p \in \mathcal{F}$
8 **upon** *apl* **response** $deliver(p', ReadyPrep(r_s, b))$
9 $R \leftarrow R \cup \{deliver(p', ReadyPrep(r_s, b))\}$
10 **if** $\exists max(b') = b_{max}$ $s.t. MaxReadiedPrep < b'$ $\exists blocking(\mathbf{self}) = bs.\ \forall u \in bs,$ $\exists deliver(u, ReadyPrep(r_u, b_u)) \in R \land (b'' \lnsim b_u \lor b''.r <= r_u)$ *for every* $b'' \lnsim b'$ **then**
11 **if** $MaxReadiedPrep = \langle r_m, v_m \rangle \land v_m \neq b_{max}.v$ **then**
12 $SecondReadiedPrep \leftarrow r_m$
13 $MaxReadiedPrep \leftarrow b_{max}$
14 *apl* **request** $send(p, ReadyPrep\ (SecondReadiedPrep, b_{max}))$ for each $p \in \mathcal{F}$
15 **if** $\exists max(b') = b_{max}$ $s.t. MaxDeliveredPrep < b'$ $\exists q \in Q.\ \forall u \in q, \exists deliver(u, ReadyPrep(b_u)) \in R \land (b'' \lnsim b_u \lor b''.r <= r_u)$ *for every* $b'' \lnsim b'$ **then**
16 $MaxDeliveredPrep \leftarrow b_{max}$
17 **response** $prepared(MaxDeliveredPrep)$

Algorithm 5: Bunched voting (BV) commit (1/2)

1 **Implements:** Bunchedvoting
2 **request** : $commit(b)$
3 **response** : $committed(b)$
4 **Vars:**
5 Q ▷ Minimal quorums of **self**
6 $\mathcal{F}$: Set[$\mathcal{P}$] ▷ The followers of **self**
7 $BEchoedCmt, BReadiedCmt, BDeliveredCmt : \langle \mathbb{N}^+, V \rangle \leftarrow \langle 0, \bot \rangle$
8 $EC, RC : messages \leftarrow \emptyset$
▷ Set of EchoCmt and ReadyCmt messages
9 **Uses:**
10 *apl* : PointToPointLink
11 **upon request** $commit(b)$ from sender
12 *apl* **request** $send(p, Commit(b))$ for each $p \in \mathcal{P}$
13 **upon** *apl* **response** $deliver(p', Commit(b))$
14 **if** $MaxDeliveredPrep = b$ **then**
15 $BEchoedCmt \leftarrow b$ *apl* **request** $send(p, EchoCmt(b))$ for each $p \in \mathcal{F}$

Algorithm 6: Bunched voting (BV) commit (2/2)

```
1  upon apl response deliver(p′, EchoCmt(b))
2      EC ← EC ∪ {deliver(p′, EchoCmt(b))}
3      if ∃q ∈ 𝒬, ∀u ∈ q, ∃deliver(u, EchoCmt(b)) ∈ EC then
4          BReadiedCmt ← b apl request send(p, ReadyCmt(b)) for each p ∈ ℱ
5  upon apl response deliver(p′, ReadyCmt(b))
6      RC ← RC ∪ {deliver(p′, ReadyCmt(b))}
7      if b ≠ BReadiedCmt ∧ ∃blocking(self) = bs, ∀u ∈ bs, ∃deliver(u, ReadyCmt(b)) ∈ RS
        then
8          BReadiedCmt ← b
9          apl request send(p, ReadyCmt(b)) for each p ∈ ℱ
10     if b ≠ BDeliveredCmt ∧ ∃q ∈ 𝒬, ∀u ∈ q, ∃deliver(u, ReadyCmt(b)) ∈ RC  then
11         BDeliveredCmt ← b
12         response committed(b)
```

Algorithm 7: Practical Byzantine Consensus (1/2)

```
1  Implements: Consensus
2    request : propose(v)
3    response : decide(v)
4  Vars:
5    round : ℕ⁺ ← 0                                    ▷ Current round number
6    candidate, prepared : ⟨ℕ⁺, V⟩ ← ⟨0, ⊥⟩
7    leader : 𝒫 ← p_0                                  ▷ current leader
8  Uses:
9    bv : BunchedVoting
10   le : EventualLeaderElection
11   apl : PointToPointLink
12 upon request propose(v)
13     candidate ← ⟨1, v⟩
14     if self = leader then
15         bv request prepare(candidate)
16 upon bv response prepared(b) where prepared < b
17     prepared ← b
18     if self = leader ∧ prepared = candidate then
19         bv request commit(candidate)
```

Algorithm 8: Practical Byzantine Consensus (2/2)

```
1  upon bv response committed(b) from p where b = prepared ∧ p = leader
2      response decide(b.v)
3  upon timeout triggered
4      le request Complain(round)
5  upon le response new-leader(p)
6      leader ← p
7      round ← round + 1
8      apl request send(p, ReadyPrep(SecondReadiedPrep, MaxReadiedPrep)) for each p ∈ F
9      if self = leader then
10         Delay for time Δ
11     start-timer(round)
12     if prepared = ⟨0, ⊥⟩ then
13         candidate ← ⟨round, candidate.v⟩
14     else
15         candidate ← ⟨round, prepared.v⟩
16     if self = leader then
17         bv request prepare(candidate)
```

only sends and receives finite number of messages to provide liveness properties. Note that instead of a relying on the assumption of partially synchronized network, where all the messages between correct processes will be delivered eventually, our protocols guarantee progress even if messages sent before GST are dropped. Practical Byzantine consensus protocol uses an instance of bunched voting to prepare and commit ballots, which generalizes and contains potentially infinite instances of Byzantine reliable broadcast in Alg. 1. *prepare*(b) is used to abort any ballots that is below and incompatible with b. *commit*(b) aims to commit ballot b. In the bunched voting and practical Byzantine consensus, we redefine the below and incompatible relation between two ballots b and b': $b \lesssim b', b = \langle r, v\rangle, b' = \langle r', v'\rangle$ iff and only if $r < r' \wedge v \neq v'$.

Alg. 3 introduces bunched voting. Each process p stores a set of its individual minimal quorums Q, and its set of followers $\mathcal{F}$. It also stores the highest ballot for which p has echoed, readied or delivered a prepare statements in *MaxEchoedPrep*, *MaxReadiedPrep* and *MaxDeliveredPrep*. It stores the highest ballots for which p has echoed, readied and

delivered a commit statement in *BEchoedCmt*, *BReadiedCmt* and *BDeliveredCmt*. All the echo and ready messages for prepare and commit statements are stored in E, R, EC, RC and wait to be processed when a quorum or blocking set of consistent messages are received. After the fields of prepare and commit statements has advance to b, we can safely discard messages with ballots smaller and incompatible with b in E, R, EC, RC. Each process also keeps the second highest round number where it has sent *ReadyPrep* message for a ballot b and $b \lnapprox MaxReadiedPrep$.

Upon receiving a request to prepare a ballot b (at line 12), the sender broadcast the ballot b to all the processes (at line 13). Upon receiving the message from the sender (at line 14), a correct process echoes the message among its followers (at line 16) only if two conditions are satisfied: it has not sent $EchoCmt(b')$ messages for any ballot $b' \lnapprox b$; b is greater than any ballot b'' that it has previously sent $EchoPrep(b'')$. The protocol proceed with the similar stages of Byzantine reliable broadcast with the following modifications: for prepare statements, at each stage the protocol only sends or delivers the maximum ballot among the ballots which are allowed. For example, when a correct process p receives a $EchoPrep(b)$ message (at line 1), it checks whether there is a ballot b' that is greater than *MaxReadiedPrep*, and for a quorum q of p, p has received an $EchoPrep(b_u)$ message from each member u of q, such that for all the ballots $b'' \lnapprox b', b'' \lnapprox b_u$. The correct process p then select the maximum ballot b_{max} among the ballots b' and update *SecondReadiedPrep* (at line 5) and *MaxReadiedPrep* (at line 6). *SecondReadiedPrep* is used to keep the highest round number of a ballot b' such that p has sent $ReadyPrep(r', b')$ message previously and b' carries a different value compared to b_{max}. It is safe to express abort intention on behalf of

p for any ballot with a round number smaller or equal to such *SecondReadiedPrep*. Because p previously agreed to abort any ballot $b_s \lessapprox \langle SecondReadiedPrep, v_s \rangle$ and now agrees to abort any ballot $b_m \lessapprox b_{max}$. Then if $v_s \neq b_{max}.v$, p agrees to abort all ballots up to rounds *SecondReadiedPrep*. For the ballots with a round number between *SecondReadiedPrep* and $b_{max}.r$, p agrees to abort any ballot with a different value compared to $b_{max}.v$. With this intuition, the correct process sends $ReadyPrep(SecondReadiedPrep, b_{max})$ to its follows (at line 7). When a correct process receives $ReadyPrep(r_s, b)$ message (at line 8), it checks similar conditions for blocking set (at line 10) and quorum (at line 15) to amplify (at line 14) and deliver (at line 17) the prepare statement.

The commit statement follows directly from the Byzantine reliable broadcast with one modification: correct process only sends $EchoCmt(b)$ for ballot $b = MaxDeliveredPrep$ (at line 14). It means a correct process always prepares a ballot first then tries to commit it.

The practical Byzantine consensus protocol (Alg. 7) follows the same structure in Alg. 2 with one caveat in order to progress: all the correct processes resend *ReadyPrep* messages at the beginning of each epoch (at line 8). This ensures that even if messages before GST are dropped, in the first round after GST, all the correct processes can synchronize their prepared ballot and a correct leader will adopt the highest prepared ballot as proposal.

3.7 Example Execution for Consensus

We demonstrate an execution of our consensus protocol, which illustrates its ability to fulfill both safety and liveness properties. Let us assume $\mathcal{P} = \{1,2,3,4\}, \mathcal{B} = \{2\}, \mathcal{Q}(1) = \{\{1,2,3\}\}, \mathcal{Q}(3) = \{\{3,4\},\{1,3\}\}, \mathcal{Q}(4) = \{\{3,4\}\}$. All the well-behaved processes are initialized with a default ballot $\langle 0, \perp \rangle$ for their *prepared* and *candidate* variable. The well-behaved processes propose three different values: process 1 propose 3; process 3 propose 5; process 4 propose 2.

In the first round, process 1 is the leader. Upon the proposal at line 11, all the well-behaved process update their *candidate* with their own proposal value at line 12. Process 1 aborts all the ballots below and incompatible with its *candidate* through reliable broadcast: it *BCast*$\mathbb{A}$ for instances $\langle 0, \perp \rangle$, $\langle 1, 1 \rangle$ and $\langle 1, 2 \rangle$ at line 14. Since the network is partially synchronized, we let the process 4 be partitioned from the rest of processes temporarily. Byzantine process 2 only send *Echo*$\mathbb{A}$ messages for ballot $\langle 0, \perp \rangle$ and $\langle 1, 1 \rangle$. Since process 1 has the individual minimal quorum $\{1,2,3\}$, it send *Ready*$\mathbb{A}$ messages for the aforementioned two ballots. Process 3 sends *Ready*$\mathbb{A}$ messages for the two ballots, too. We partition process 3 from process 1 and 2 from now on temporarily. Process 1 is able to deliver the $\mathbb{A}$ for ballots $\langle 0, \perp \rangle$ and $\langle 1, 1 \rangle$ through BRB at line 15 and update its *prepared* variable as $\langle 1, 2 \rangle$ at line 16. Since the leader process 1's $\textit{prepared} \neq \textit{candidate}$, it can not broadcast commit for its *candidate* and the timer triggers.

In the second round, Byzantine process 2 is the leader. When the new leader is elected for a new round, all the well-behaved process increase their round number and update their *candidate*: process 1 updates its *candidate* value to its prepared value 2 at

line 34; process 3 and 4 has not prepared any ballot and update their candidate to their original proposal at line 31. Then Byzantine leader tries to commit an arbitrary value 4, which no well-behaved process will accept, since the well-behaved process only deliver $\mathbb{C}$ for a BRB instance with the ballot same as its *prepared* at line 19. We recovery the connection from process 3 and 4 to the rest of the network. Then both process 3 and 4 delivers $\mathbb{A}$ message from previous round at line 15 and update their *prepared* ballot to $\langle 1, 2\rangle$. No value is decided and the timer trigger again.

In the third round, well-behaved process 3 is the leader. All the well-behaved process now synchronized to the same highest prepared ballot $\langle 1, 2\rangle$ and update their *candidate* accordingly. Process 3 then abort the rest ballots that are less and incompatible with its $\mathit{candidate} = \langle 3, 2\rangle$ through BRB. Byzantine process 2 remain silent in this round to prevent liveness for the weakly available process 1. However, process 3 and 4 are strongly available and is able to successfully deliver the abort messages. Their *prepared* is updated to $\langle 3, 2\rangle$ and process 3 commits this ballot through BRB at line 18. All the process 3 and 4 deliver the commit message at line 19 and decide the value 2 in the committed ballot $\langle 3, 2\rangle$ at line 20.

1	2	3	4
$p = \langle 0, \perp \rangle$ $c = \langle 0, \perp \rangle$		$p = \langle 0, \perp \rangle$ $c = \langle 0, \perp \rangle$	$p = \langle 0, \perp \rangle$ $c = \langle 0, \perp \rangle$
Propose(*3*)		*Propose*(*5*)	*Propose*(*2*)
$p = \langle 0, \perp \rangle$ $c = \langle 1, 3 \rangle$		$p = \langle 0, \perp \rangle$ $c = \langle 1, 5 \rangle$	$p = \langle 0, \perp \rangle$ $c = \langle 1, 2 \rangle$
$BCast(\langle 0, \perp \rangle, Abort)$ $BCast(\langle 1, 1 \rangle, Abort)$ $BCast(\langle 1, 2 \rangle, Abort)$			
$Echo(\langle 0, \perp \rangle, Abort)$ $Echo(\langle 1, 1 \rangle, Abort)$ $Echo(\langle 1, 2 \rangle, Abort)$	$Echo(\langle 0, \perp \rangle, Abort)$ $Echo(\langle 1, 1 \rangle, Abort)$	$Echo(\langle 0, \perp \rangle, Abort)$ $Echo(\langle 1, 1 \rangle, Abort)$ $Echo(\langle 1, 2 \rangle, Abort)$	slow
$Ready(\langle 0, \perp \rangle, Abort)$ $Ready(\langle 1, 1 \rangle, Abort)$	$Ready(\langle 0, \perp \rangle, Abort)$ $Ready(\langle 1, 1 \rangle, Abort)$	$Ready(\langle 0, \perp \rangle, Abort)$ $Ready(\langle 1, 1 \rangle, Abort)$	slow
$Deliver(\langle 0, \perp \rangle, Abort)$ $Deliver(\langle 1, 1 \rangle, Abort)$		slow	slow
$p = \langle 1, 2 \rangle$ $c = \langle 1, 3 \rangle$		$p = \langle 0, \perp \rangle$ $c = \langle 0, \perp \rangle$	$p = \langle 0, \perp \rangle$ $c = \langle 0, \perp \rangle$
time out		time out	time out
$p = \langle 1, 2 \rangle$ $c = \langle 2, 2 \rangle$		$p = \langle 0, \perp \rangle$ $c = \langle 2, 5 \rangle$	$p = \langle 0, \perp \rangle$ $c = \langle 2, 2 \rangle$
	$BCast(\langle 2, 4 \rangle, Commit)$		
		$Deliver(\langle 0, \perp \rangle, Abort)$ $Deliver(\langle 1, 1 \rangle, Abort)$	$Echo(\langle 0, \perp \rangle, Abort)$ $Echo(\langle 1, 1 \rangle, Abort)$ $Echo(\langle 1, 2 \rangle, Abort)$ $Ready(\langle 0, \perp \rangle, Abort)$ $Ready(\langle 1, 1 \rangle, Abort)$ $Delivery(\langle 0, \perp \rangle, Abort)$ $Delivery(\langle 1, 1 \rangle, Abort)$
$p = \langle 1, 2 \rangle$ $c = \langle 2, 2 \rangle$		$p = \langle 1, 2 \rangle$ $c = \langle 2, 5 \rangle$	$p = \langle 1, 2 \rangle$ $c = \langle 2, 2 \rangle$
time out		time out	time out
$p = \langle 1, 2 \rangle$ $c = \langle 3, 2 \rangle$		$p = \langle 1, 2 \rangle$ $c = \langle 3, 2 \rangle$	$p = \langle 1, 2 \rangle$ $c = \langle 3, 2 \rangle$
		$BCast(\langle 1, 2 \rangle, Abort)$ ...	
	silent	$Deliver(\langle 1, 2 \rangle, Abort)$...	$Deliver(\langle 1, 2 \rangle, Abort)$...
$p = \langle 1, 2 \rangle$ $c = \langle 3, 2 \rangle$		$p = \langle 3, 2 \rangle$ $c = \langle 3, 2 \rangle$	$p = \langle 3, 2 \rangle$ $c = \langle 3, 2 \rangle$
		$BCast(\langle 3, 2 \rangle, Commit)$	
	silent	$Deliver(\langle 3, 2 \rangle, Commit)$	$Deliver(\langle 3, 2 \rangle, Commit)$
	silent	$Decide(2)$	$Decide(2)$

Table 3.2: An execution for consensus protocol with leader switch

3.8 Discussion

When Byzantine processes don't lie about their slices, an FBQS enjoys quorum sharing [311]. Consider a quorum q of a process p, and a process p' in it. Since a set is recognized as a quorum only if it contains a slice for each of its members, the process p' has a slice s in q. Processes receive the same set of slices from a given process even if it is Byzantine. If p' starts from s, it can find a quorum that grows no larger than q, *i.e.*, it can stop at q if not earlier. Therefore, q includes a quorum of p'.

However, when Byzantine processes lie about their slices, FBQS does not satisfy quorum sharing. Consider an FBQS with five processes $\mathcal{P} = \{1, 2, 3, 4, 5\}$, and one Byzantine process $\mathcal{B} = \{4\}$. The slices of well-behaved processes $\mathcal{W} = \{1, 2, 3, 5\}$ are the following: $\mathcal{S}(1) = \{\{1, 2\}\}$, $\mathcal{S}(2) = \{\{2, 3\}, \{2, 4\}, \{2, 5\}\}$, $\mathcal{S}(3) = \{\{3, 4\}\}$, $\mathcal{S}(5) = \{\{2, 5\}\}$. However, the Byzantine process 4 provides different quorum slices to different well-behaved processes. The slices that it sends to each other process are the following: $\mathcal{S}(4)_1 = \{\{1, 4\}\}$, $\mathcal{S}(4)_2 = \{\{3, 4\}\}$, $\mathcal{S}(4)_3 = \{\{2, 4\}\}$, $\mathcal{S}(4)_5 = \{\{4, 5\}\}$. By the definition of quorums in FBQS, we have the following individual minimal quorums for each well-behaved process: $\mathcal{Q}(1) = \{\{1, 2, 4\}, \{1, 2, 5\}\}$, $\mathcal{Q}(2) = \{\{2, 3, 4\}, \{2, 5\}\}$, $\mathcal{Q}(3) = \{\{2, 3, 4\}\}$, $\mathcal{Q}(5) = \{\{2, 5\}\}$. We can see that $\mathcal{Q}$ does not have quorum sharing, since the quorum $\{1, 2, 4\}$ does not include any quorum for process 2. However, $\mathcal{Q}$ is strongly available for processes $\{1, 2, 5\}$, since they all have quorums that are well-behaved and quorum subsuming. Notice that abstract quorums which are over-approximations of FBQS satisfy quorum-sharing [311]: A set q is an abstract quorum of a process p if p is Byzantine, or p has a slice contained in q and every well-behaved member of q has a slice contained in q. However, as shown in

our example, not all abstract quorums of a process can be observed by that process when Byzantine processes lie about their quorum slices. For example, $\{1, 2, 4\}$ is an abstract quorum of process 2 but 2 does not observe it when the Byzantine process 4 lies. Therefore, relaxation of the quorum sharing is necessary when analyzing quorum systems such as FBQS when Byzantine processes lie.

Further, there are systems where a strongly available set exists but no guild set (from ABQS) exists. Therefore, HQS can provide safety and liveness for those executions but ABQS cannot. We show a quorum system that has an empty guild but has non-empty strongly available set and quorum intersection. Let $\mathcal{P} = \{1, 2, 3, 4\}$, and process 3 is Byzantine. Let the asymmetric failure prone system be $\mathcal{F}(1) = \{\{3, 4\}, \{2\}\}$, $\mathcal{F}(2) = \{\{3, 4\}, \{1\}\}$, $\mathcal{F}(3) = \{\{1, 2\}\}$, $\mathcal{F}(4) = \{\{2, 3\}\}$. The canonical quorum systems are $\mathcal{Q}(1) = \{\{1, 2\}, \{1, 3, 4\}\}$, $\mathcal{Q}(2) = \{\{1, 2\}, \{2, 3, 4\}\}$, $\mathcal{Q}(3) = \{\{3, 4\}\}, \mathcal{Q}(4) = \{\{1, 4\}\}$. This HQS has quorum intersection and a non-empty strongly available set: all the quorums of well-behaved processes have at least one correct process in their intersection. The set $\{1, 2\}$ is a strongly available set for $\mathcal{Q}$. However, it is not an ABQS with generalized B^3 condition. For processes 1 and 2, $\mathcal{F}_1 = \{2\}, \mathcal{F}_2 = \{1\}$ and $\mathcal{F}_{12} = \{3, 4\}$, and we have $\mathcal{P} \subseteq \{2\} \cup \{1\} \cup \{3, 4\}$, which violates generalized B^3. Therefore this quorum system is not an ABQS. There is no guild set.

3.9 Related Works

Quorum Systems with Heterogeneous Trust. Ripple [397] and Cobalt [319] pioneered decentralized trust. They let each node specify a list, called the unique node list

(UNL), of processes that it trusts. However, they do not consider quorum availability or subsumption.

Stellar [334, 307] presents federated Byzantine quorum systems (FBQS) [185, 187] where quorums are iteratively calculated from quorums slices. Stellar also presents a federated voting and consensus protocol. In comparison, the assumptions of the protocols presented in this paper are weaker, and their guarantees are stronger. The stellar consensus protocol (SCP) guarantees termination when Byzantine processes stop. In contrast, the consensus protocol in this paper guarantees termination regardless of Byzantine processes. Further, abstract SCP [185] provides agreement only for intact processes. The intact set for an FBQS is a subset of processes that have strong availability. On the other hand, the consensus protocol in this paper provides agreement for all well-behaved processes. In FBQS, the intersections of quorums should have a process in the intact set; however, in HQS, they only need to have a well-behaved process. The validity and totality properties for the reliable broadcast for FBQS are restricted to the intact set. On the other hand, the reliable broadcast protocol in this paper provides totality for all processes that have weak availability, and validity for all processes that have strong availability.

Personal Byzantine quorum systems (PBQS) [311] capture the quorum systems that FBQSs derive form slices, and propose a responsiveness consensus protocol [445, 17, 367, 24]. They define a notion called quorum sharing which requires quorum subsumption for every quorum. Stellar quorums have quorum sharing if and only if processes do not lie about their slices. (The appendix [290] presents examples.) In this paper, we relax quorum sharing to quorum subsumption, and capture quorums that FBQSs derive even

when Byzantine quorums lie about their slices, and show that even if a quorum system does not satisfy quorum sharing, safety can be maintained for all processes, and liveness can be maintained for the set of strongly available processes.

Asymmetric Byzantine quorum systems (ABQS) [104, 105, 25] allow each process to define a subjective dissemination quorum system (DQS), in a globally known system. The followup model [103] lets each process specify a subjective DQS for processes that it knows, transitively relying on the assumptions of other processes. In contrast, HQS lets each process specify its own set of quorums without knowing the quorums of other processes. Further, it does not require the specification of a set of possible Byzantine sets. Further, there are systems where a strongly available set (from HQS) exists but no guild set (from ABQS) exists. (The appendix [290] presents examples.) Therefore, HQS can provide safety and liveness for those executions but ABQS cannot. ABQS presents shared memory and broadcast protocols, and further, rules to compose two ABQSs. On the other hand, this paper proves impossibility results, and presents protocols for reliable broadcast and consensus abstractions. HQS provides strictly stronger guarantees with weaker assumptions. In ABQS, the properties of reliable broadcast are stated for wise processes and the guild. However, this paper states these four properties for well-behaved processes and the strongly available set. Well-behaved processes are a superset of wise processes, and as noted above, in certain executions, the strongly available set is a superset of the guild.

Flexible BFT [325] allows different failure thresholds between learners. Heterogeneous Paxos [401, 402] further generalizes the separation between learners and acceptors with different trust assumptions; it specifies quorums as sets rather than number of pro-

cesses. These two projects introduce a consensus protocol that guarantees safety or liveness for learners with correct trust assumptions. However, they require the knowledge of all processes in the system. In contrast, HQS only requires partial knowledge of the system, and captures the properties of quorum systems where reliable broadcast and consensus protocols are impossible or possible. Multi-threshold reliable broadcast and consensus [223] and MT-BFT [341] elaborate Bracha [86] to have different fault thresholds for different properties, and different synchrony assumptions. However, they have cardinality-based or uniform quorums across processes. In contrast, HQS supports heterogeneous quorums.

K-consistent reliable broadcast (K-CRB) [70] introduces a relaxed reliable broadcast abstraction where the correct processes can define their own quorum systems. Given a quorum system, it focuses on delivering the smallest number k of different values. In contrast, we propose the weakest condition to solve classical reliable broadcast and consensus. Moreover, K-CRB's relaxed liveness guarantee (accountability) requires public key infrastructure. In contrast, all the results in this paper are for information-theoretic setting.

Our consensus protocol uses eventual leader election. Several other works present view synchronization and eventual leader election for Byzantine replicated systems [88, 87], and dynamic networks [344, 229]. It is interesting to see if their leader election modules can be generalized to the heterogeneous setting, and support responsiveness [445, 40] for our consensus protocol.

Impossibility Results. There are two categories of assumptions about the computational power of Byzantine processes. In the information-theoretic setting, Byzantine process have unlimited computational resources. While in the computational setting,

Byzantine processes can not break a polynomial-time bound [183]. In this work, our impossibility results for reliable broadcast and consensus fall in the information-theoretic category. Whether the same results hold in the computational setting is an interesting open question.

FLP [172] proved that consensus is not solvable in asynchronous networks even with one crash failure. Many following works [198, 146, 20, 171, 267, 83] considered solvability, and necessary and sufficient conditions for consensus and reliable broadcast to tolerate f Byzantine failures in partially synchronous networks. The number of processes should be more than $3f$ and the connectivity of the communication graph should be more than $2f$. However, these results apply for cardinality-based quorums, which is a special instance of HQS. We generalize the reliable broadcast and consensus abstractions to HQS which supports non-uniform quorums, and prove impossibility results for them.

3.10 Conclusion

This paper presented a general model of heterogeneous quorum systems where each process defines its own set of quorums, and captured their properties. Through indistinguishably arguments, it proved that no deterministic quorum-based protocol can implement the consensus and Byzantine reliable broadcast abstractions on a heterogeneous quorum system that provides only quorum intersection and availability. It introduced the quorum subsumption property, and showed that the three conditions together are sufficient to implement the two abstractions. It presented Byzantine broadcast and consensus protocols for heterogeneous quorum systems, and proved their correctness when the underlying quorum system maintain the three properties.

Chapter 4

Reconfigurable Heterogeneous Quorum Systems

4.1 Introduction

Banks have been traditionally closed; only established institutions could hold accounts and execute transactions. With regulations in place, this centralized model can preserve the integrity of transactions. However, it makes transactions across these institutions costly and slow; further, it keeps the power in the hands of a few. In pursuit of decentralization, Bitcoin [350] provided open membership: any node can join the Bitcoin network, and validate and process transactions. It maintains a consistent replication of an append-only ledger, called the blockchain, on a dynamic set of global hosts including potentially malicious ones. However, it suffers from a few drawbacks: low throughput, high energy consumption, and only probabilistic guarantees of commitment [281, 282].

Maintaining consistent replication in the presence of malicious processes has been the topic of Byzantine replicated systems for decades. PBFT [116] and its numerous following variants [423, 338, 445, 410, 35, 411] can maintain consistent replication when the network size is at least three times the size of potentially Byzantine coalitions, have higher throughput than Bitcoin, have modest energy consumption, give participants equal power, and provide deterministic liveness guarantees. Unfortunately, however, their quorums are uniform and their membership is closed. Their trust preferences, *i.e.*, the quorums of processes are fixed and homogeneous across the network. Further, their set of participants are fixed; thus, in contrast to proof-of-work replication that provides permissionless blockchains, classical Byzantine replication only provides permissioned blockchains.

Can the best of both worlds come together? Can we keep the consistency, throughput, modest energy consumption and equity of Byzantine replicated systems, and bring heterogeneous trust [142, 105, 25] and *open membership* to it? Openness challenges classical assumptions. With global information about the processes and their quorums, classical quorum systems could be configured at the outset to satisfy consistency and availability properties. However, open quorum systems relinquish global information as processes specify their own quorums, and can further join, leave, and reconfigure their quorums. As the other processes may be unaware of these changes, consistency and availability may be violated after and even while these reconfigurations happen.

Projects such as Ripple [397] and Stellar [334] pioneered, and follow-up research [311, 307, 187, 85] moved towards this goal, and presented quorum systems where nodes can specify their own quorums, and can join and leave. In fact, the Stellar network has

a high churn. In previous works, the consistency of the network is either assumed to be maintained by user preferences or a structured hierarchy of nodes, is provided only in divided clusters of processes, or can be temporarily violated and is periodically checked across the network. Reconfigurations can compromise the consistency or availability of the replicated system. The loss of consistency can be the antecedent to a fork and double-spending. An important open problem is *reconfiguration protocols for heterogeneous quorum systems with provable security guarantees*. The protocols are expected to avoid external central oracles, or downtime.

In this paper, we first present a *general model of heterogeneous quorum systems* where each process declares its individual set of quorums, and then formally capture the properties of these systems: consistency, availability and inclusion. We then consider the *reconfiguration* of heterogeneous quorum systems: joining and leaving of a process, and adding and removing of a quorum. To cater for the protocols such as broadcast and consensus that use the quorum system, the reconfiguration protocols are expected to preserve the above properties.

The safety of consensus naturally relies on the *consistency (or quorum intersection)* property: every pair of quorums intersect at a well-behaved process. Intuitively, if an operation communicates with a quorum, and a later operation communicates with another quorum, only a well-behaved process in their intersection can make the second aware of the first. A quorum system is *available* for a process if it has a well-behaved quorum for that process. Intuitively, the quorum system is responsive to that process through that quorum. The less known property is *quorum inclusion*. Roughly speaking, every quorum should

include a quorum of each of its members. This property trivially holds for homogeneous quorum systems where every quorum is uniformly a quorum of all its members, but should be explicitly maintained for heterogeneous quorum systems. We show that quorum inclusion interestingly lets processes in the included quorum make local decisions while preserving properties of the including quorum. We precisely capture and illustrate these properties.

We then present *quorum graphs*, a graph characterization of heterogeneous quorum systems with the above properties. It is known that strongly connected components of a graph form a directed acyclic graph (DAG). We prove that a quorum graph has only one *sink component*, and preserving consistency reduces to preserving quorum intersections in this component. This fact has an important implication for optimization of reconfiguration protocols. Any change outside the sink component preserves consistency, and therefore, can avoid synchronization with other processes. Thus, we present a decentralized *sink discovery protocol* that can find whether a process is in the sink.

In addition to consistency, availability and inclusion, reconfiguration protocols are expected to preserve *policies*. Each process declares its own trust policy: it specifies the quorums that it trusts. In particular, it does not trust strict subsets of its individual quorums. Thus, a policy-preserving reconfiguration should not shrink any quorum. We present a *join protocol* that preserves all the above properties. We present *trade-offs* for the properties that the leave, remove and add reconfiguration protocols can preserve. We show that there is no *leave or remove protocol* that can preserve both the policies and availability. Thus, we present two protocols: a protocol that preserves policies, and another that preserves availability. Both preserve consistency and inclusion. Then, we show that

there is no *add protocol* that can preserve both the policies and consistency. Therefore, since we never sacrifice consistency, we present a protocol that preserves all properties except the policies.

We observe that under reconfiguration, *quorum inclusion is critical* to preserve not only availability but also consistency. Sometimes, reconfigurations can only eventually reconstruct inclusion, but can preserve *weaker notions of inclusion* that are sufficient to preserve consistency and availability. We capture these notions, prove that they are preserved, and use them to prove that the other properties are preserved.

4.2 Quorum Systems

Processes. A quorum system is hosted on a set of processes $\mathcal{P}$. In each execution, $\mathcal{P}$ is partitioned into *Byzantine* $\mathcal{B}$ and *well-behaved* $\mathcal{W} = \mathcal{P} \setminus \mathcal{B}$ processes. Well-behaved processes follow the given protocols; however, Byzantine processes can deviate from the protocols arbitrarily. Furthermore, a well-behaved process does not know the set of well-behaved processes $\mathcal{W}$ or Byzantine processes $\mathcal{B}$. The active processes $\mathcal{A} \subseteq \mathcal{P}$ are the current members of the system. As we will see in section 4.5, quorum systems can be reconfigured, and the active set can change: processes can join and the active set grows, and conversely, processes can leave, and the active set shrinks.

We consider partially synchronous networks [160], *i.e.*, if both the sender and receiver are well-behaved, the message will be eventually delivered within a bounded delay after an unknown GST (Global stabilization Time). Processes can exchange messages on authenticated point-to-point links.

$$
\begin{aligned}
&\mathcal{P} = \mathcal{W} \cup \mathcal{B}, \quad \mathcal{W} = \{1,2,3,5\}, \quad \mathcal{B} = \{4\}\\
&\mathcal{Q} = \{1 \mapsto \{\{1,2,4\}\},\\
&\qquad 2 \mapsto \{\{1,2\},\{2,3\},\{2,5\}\},\\
&\qquad 3 \mapsto \{\{2,3\}\},\\
&\qquad 5 \mapsto \{\{2,5\}\}\}
\end{aligned}
$$

Figure 4.1: Example Quorum System

Individual Quorums. Processes can have different trust assumptions: trust is a subjective matter, and therefore, heterogeneous. We capture a heterogeneous model of quorum systems where each process can specify its individual set of quorums.

An *individual quorum* q of a process p is a non-empty subset of processes in $\mathcal{P}$ that p trusts to collectively perform an operation. Every quorum of a process p naturally contains p itself. (However, this is not necessary for any theorem in this paper.) By the above definition, any superset of a quorum of p is also a quorum of p. Thus, the set of quorums of p is superset-closed and has minimal members. (Consider a set of sets $S = \{\overline{s}\}$. We say that S is superset-closed, if any superset s' of any member s of S is a member of S as well.) For example, let the minimal quorums of process 1 be the set $\{\{1,4\},\{1,3\}\}$. Then, the set $\{1,3,4\}$ is a quorum of 1 but is not a minimal quorum of 1. A process p doesn't need to keep any quorum other than its minimal quorums: any of its other quorums include extra processes that p can perform operations without. Thus, we consider only the *(individual) minimal quorums* of p. Any superset of such a quorum is a *quorum* for p. We denote a set of quorums as Q. We denote the union of a set of quorums Q as $\cup Q$.

Heterogeneous Quorum Systems. In a heterogeneous quorum system, the set of individual quorums can be different across processes.

Definition 68 (Quorum System) *A heterogeneous quorum system (HQS) $\mathcal{Q}$ maps each active process to a non-empty set of individual minimal quorums.*

The mapping models the fact that each process has only a local view of its own individual minimal quorums. Consider the running example in Figure 4.1. The minimal quorums of process 2 are $\{1, 2\}$, $\{2, 3\}$ and $\{2, 5\}$. Further, since the behavior of Byzantine processes can be arbitrary, we leave their individual quorums unspecified.

When obvious from the context, we say quorum systems to refer to heterogeneous quorum systems, and say quorums of p to concisely refer to the individual minimal quorums of p.

Quorums. Next, we consider quorums and their minimality across all processes of a quorum system. Consider a quorum system $\mathcal{Q}$. The set of (individual) quorums of $\mathcal{Q}$ is the set of quorums in the range of the map $\mathcal{Q}$. A quorum q is a *minimal quorum* of $\mathcal{Q}$ iff q is an individual minimal quorum of a process in $\mathcal{Q}$, and no proper subset of q is an individual minimal quorum of any process in $\mathcal{Q}$. (A minimal quorum is also called elementary [311].) We denote the set of minimal quorums of $\mathcal{Q}$ as $MQ(\mathcal{Q})$. In our running example in Figure 4.1, $MQ(\mathcal{Q}) = \{\{1, 2\}, \{2, 3\}, \{2, 5\}\}$. We note that although $\{1, 2, 4\}$ is a minimal quorum of 1, it is not a minimal quorum of $\mathcal{Q}$ since since 2 has the quorum $\{1, 2\}$ that is a strict subset of $\{1, 2, 4\}$.

Lemma 69 *For all quorum systems $\mathcal{Q}$, every minimal quorum of $\mathcal{Q}$ is an individual minimal quorum of some process in $\mathcal{Q}$. Further, every quorum of $\mathcal{Q}$ is a superset of a minimal quorum of $\mathcal{Q}$.*

4.3 Properties

The *consistency, availability and inclusion* properties are expected to be provided by a quorum system, and maintained by a reconfiguration protocol. In this section, we precisely define these notions. We adapt consistency and availability for HQS, and define the new notion of inclusion. We then consider a few variants of HQS. The conditions are parametric for a Byzantine attack, *i.e.*, the set of Byzantine processes $\mathcal{B}$ (or equivalently the set of well-behaved processes $\mathcal{W}$). Each condition can be directly lifted for a set of attacks $\{\overline{\mathcal{B}}\}$ by requiring the condition for each $\mathcal{B}$.

Consistency. A process stores and retrieves information from the quorum system by communicating with one of its quorums. Therefore, to ensure that each operation observes the previous one, the quorum system is expected to maintain an intersection for any pair of quorums at well-behaved processes. A set of quorums have quorum intersection at a set of well-behaved processes $P \subseteq \mathcal{W}$ iff every pair of them intersect in at least one process in P.

Definition 70 (Consistency, Quorum Intersection) *A quorum system $\mathcal{Q}$ is consistent (i.e., has quorum intersection) at a set of well-behaved processes P iff the quorums of well-behaved processes have quorum intersection at P, i.e., $\forall p, p' \in \mathcal{W}.\ \forall q \in \mathcal{Q}(p),\ q' \in \mathcal{Q}(p').\ q \cap q' \cap P \neq \emptyset$.*

The set P is often implicitly the set of all well-behaved processes $\mathcal{W}$.

For example, in Figure 4.1, the quorum system $\mathcal{Q}$ is consistent since any two quorums have a well-behaved process (either 1 or 2) in their intersection. It is straightforward that every minimal quorum of a consistent quorum system contains a well-behaved process.

Lemma 71 *In every quorum system, minimal quorums have quorum intersection iff individual minimal quorums have quorum intersection.*

Immediate from Theorem 69. This has an important implication for preservation of consistency.

Lemma 72 *Every quorum system is consistent if its minimal quorums have quorum intersection.*

Straightforward from Theorem 70 and Theorem 71.

Availability. To support progress for a process, the quorum system is expected to provide at least one responsive quorum for that process.

Definition 73 (Availability) *A quorum system is available for processes P at a set of well-behaved processes P' iff every process in P has at least a quorum that is a subset of P'.*

We say that a quorum system is available for P iff it is available for P at the set of active well-behaved processes. In our running example in Figure 4.1, the quorum system $\mathcal{Q}$ is available for $\{2, 3, 5\}$ since the processes 2 and 3 have the quorum $\{2, 3\}$, process 5 has the quorum $\{2, 5\}$, and the members of the quorums, 2, 3 and 5, are well-behaved. We note that $\mathcal{Q}$ is not available for 1 since its quorum intersects Byzantine processes $\mathcal{B} = \{4\}$.

We say that a quorum system is available *inside* P iff it is available for P at P. The set P has an interesting property that we will later use to maintain consistency. Consider a process p in P. If a set of processes P' can block availability for p, then P' intersects P. In our running example in Figure 4.1, the quorum system $\mathcal{Q}$ is available inside $P = \{2, 3, 5\}$.

The set $P' = \{1, 3, 5\}$ intersects all quorums of process 2 and can block its availability. We observe that the two sets P and P' intersect.

Let's first see the notion of blocking set [311, 185] for quorums (rather than slices [334]).

Definition 74 (Blocking Set) *A set of processes P is a blocking set for a process p (or is p-blocking) iff P intersects every quorum of p.*

Lemma 75 *In every quorum system that is available inside a set of processes P, every blocking set of every process in P intersects P.*

Proof. Consider a quorum system that is available inside P, a process p in P, and a set of processes P' that blocks p. By the definition of availability, there is at least one quorum q of p that is a subset of P. By the definition of blocking, q intersects with P'. Hence, P intersects P' as well. ■

Quorum inclusion. Before defining the notion of quorum inclusion, let us start with an intuitive example of how inclusion of quorums can support their intersection. Consider a pair of quorums q_1 and q_2 that intersect at a well-behaved process p. Let a quorum q'_1 of p be included in q_1, and a quorum q'_2 of p be included in q_2. Consider that p wants to check whether it can leave without violating quorum intersection for q_1 and q_2. It is sufficient that p locally checks if there is at least one well-behaved process in the intersection of its own quorums q'_1 and q'_2.

Let us start with a simple example. The quorum system $\mathcal{Q}$ in Figure 4.1 is quorum including (for $\mathcal{W}$). For example, consider process $p = 2$, and the quorum $q = \{1, 2\}$ of p. The well-behaved processes p' of q are 1 and 2. Process 1 has the quorum $q' = \{1, 2, 4\}$

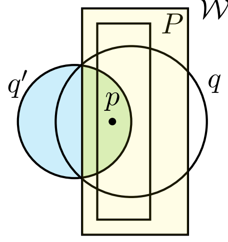


Figure 4.2: Quorum inclusion of q for P. Process p is a member of q that falls inside P, and q' is a quorum of p. Well-behaved processes of q' (shown as green) should be a subset of q.

and its well-behaved subset is $\{1, 2\}$ that is included in q. Process 2 has quorum q that is trivially a subset of itself. Figure 4.2 illustrates the following definition of quorum inclusion.

Definition 76 (Quorum inclusion) *Consider a quorum system $\mathcal{Q}$, and a subset P of its well-behaved processes.*

A quorum q is quorum including for P iff for every process p in the intersection of q and P, there is a quorum q' of p such that well-behaved processes of q' are a subset of q, i.e., $including(q, P) \coloneqq \forall p \in q \cap P.\ \exists q' \in \mathcal{Q}(p).\ q' \cap \mathcal{W} \subseteq q$.

A quorum system $\mathcal{Q}$ is quorum including for P iff every quorum of well-behaved processes of $\mathcal{Q}$ is quorum including for P, i.e., $\forall p \in \mathcal{W}.\ \forall q \in \mathcal{Q}(p).\ including(q, P)$.

The set P is often implicitly the set of all well-behaved processes $\mathcal{W}$.

Quorum inclusion was inspired by and weakens quorum sharing [311].

Definition 77 (Quorum sharing) *A quorum q has* quorum sharing *iff for every process p in q, there exists a quorum q' of p that is a subset of q. A quorum system has quorum sharing if all its quorums have quorum sharing, i.e., $\forall p, \forall q \in \mathcal{Q}(p).\ \forall p' \in q.\ \exists q' \in \mathcal{Q}(p').\ q' \subseteq q$.*

Quorum sharing requires conditions on the Byzantine processes in q and q', and is too

strong to maintain. We presented quorum inclusion that is weaker than quorum sharing. It requires a quorum q' only for well-behaved processes of q, and requires only the well-behaved subset of q' to be a subset of q. We will see in section 4.6 that quorum inclusion is sufficient to support quorum intersection.

Outlived. As we will see in our reconfiguration protocols, quorum inclusion and quorum availability support quorum intersection. Thus, we tightly integrate these three properties in the notion of *outlived quorum systems.*

Definition 78 (Outlived) *A quorum system $\mathcal{Q}$ is outlived for a set of well-behaved processes $\mathcal{O}$ iff (1) $\mathcal{Q}$ is consistent at $\mathcal{O}$, (2) available inside $\mathcal{O}$, and (3) quorum including for $\mathcal{O}$.*

In an outlived quorum system, well-behaved processes enjoy safety (quorum intersection) and outlived processes enjoy liveness (availability of a quorum with inclusion). The safety and liveness properties of outlived processes outlive Byzantine attacks, hence the name. For example, our running quorum system in Figure 4.1 is outlived for $\{2, 3, 5\}$.

We call $\mathcal{O}$ an *outlived set* for $\mathcal{Q}$, and call a member of $\mathcal{O}$ an *outlived process.* We call a quorum system that is outlived for a set, an outlived quorum system. Similarly, we use the qualifier outlived for the properties (1)-(3) above. Quorum systems are initialized to be outlived, and the reconfiguration protocols preserve this property.

HQS Instances. We now describe a few instances of HQS, and their properties.

Dissemination quorum systems (DQS). A DQS [327] (and the cardinality-based quorum system as a a special case) declares a global set of quorums for all processes. Processes have the same set of individual minimal quorums. DQS further declares a set

of possible Byzantine sets. A DQS is outlived for all well-behaved processes $\mathcal{W}$. It is consistent at $\mathcal{W}$ since the intersection of no pair of quorums falls completely in a Byzantine set. It is available for $\mathcal{W}$ since there is at least one quorum that does not intersect with any Byzantine set. It is quorum including for $\mathcal{W}$: since the quorums are global, all the well-behaved members of a quorum q recognize q as their own quorum. However, in general, an HQS may be outlived for only a subset of well-behaved processes.

Personal Byzantine quorum systems (PBQS). A PBQS [311] is an HQS that requires quorum sharing, and further quorum intersection and availability for subsets of processes called clusters. A cluster is an outlived HQS.

Federated (Byzantine) quorum systems (FBQS). An FBQS [334, 185] lets each process p specify its own quorum slices. A slice is a subset of processes that p trusts when they state the same statement. A slice is only a part of a quorum. A quorum is a set of processes that contains a slice for each of its members. A process can construct a quorum starting from one of its own slices, and iteratively probing and including a slice of each process in the set. As each process calculates its own quorums, an HQS is formed.

When Byzantine processes don't lie about their slices, the resulting HQS enjoys quorum sharing [311]. Consider a quorum q of a process p and a process p' in it. Since a set is recognized as a quorum only if it contains a slice for each of its members, there is a slice s of process p' in q. Since Byzantine processes don't lie about their slices, processes receive the same set of slices from a given process.

If p' starts from s, it can gather the same slices for the processes s as p does, and can assemble a quorum q' that grows no larger than q. Therefore, q is a superset of a

quorum q'. However, if Byzantine processes lie about their slices, quorum sharing may not hold.

4.4 Graph Characterization

We now define a graph characterization of heterogeneous quorum systems. We show that the graphs for quorum systems with certain properties have a single sink component that contains all the well-behaved processes in minimal quorums; therefore, by Theorem 72, preserving consistency reduces to preserving quorum intersections in that component.

Quorum graph. The quorum graph of a quorum system $\mathcal{Q}$ is a directed graph $G = (\mathcal{P}, E)$, where vertices are the processes, and there is an edge from p to p' if p' is a member of an individual minimal quorum of p, *i.e.*, $(p, p') \in E$ iff $\exists q \in \mathcal{Q}(p).\ p' \in q$. Intuitively, the edge (p, p') represents the fact that p directly consults with p'. For example, Figure 4.3 shows a quorum system and its graph representation. We refer to a quorum system and its graph characterization interchangeably.

We now prove a few properties for quorum systems with consistency and quorum sharing. (Quorum systems with these properties enable optimizations for reconfiguration; however, the protocols in the next sections don't require quorum sharing.)

Lemma 79 *A quorum is a minimal quorum iff it is an individual minimal quorum for all its well-behaved members.*

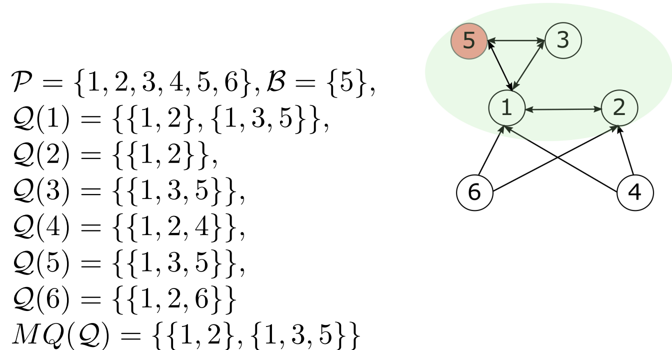


Figure 4.3: Quorum Graph Example

Proof. We first show the only-if direction. Consider a minimal quorum q. By the quorum sharing property, each well-behaved process in q has an individual minimal quorum q' such that $q' \subseteq q$. Since q is a minimal quorum, $q' = q$. The proof of the if direction is by contradiction. Assume the if condition: q is an individual minimal quorum for all its well-behaved members. However, q is not a minimal quorum. Since q is an individual minimal quorum but not a minimal quorum, by Theorem 69, there is a minimal quorum q' such that $q' \subsetneq q$. Let p be a well-behaved process in q' (and therefore, q). By the if condition, q is an individual minimal quorum of p. By the only if direction, q' is an individual minimal quorum of p. However, these two facts and $q' \subsetneq q$ contradict the minimality assumption for q. ■

We remember that a subset of vertices that are pair-wise connected are a clique.

Lemma 80 *Well-behaved processes in a minimal quorum are a clique.*

This is immediate from Theorem 79. For example, in Figure 4.3, the minimal quorums are $MQ(\mathcal{Q}) = \{\{1,2\}, \{1,3,5\}\}$, and their well-behaved processes $\{1,2\}$ and $\{1,3\}$ are cliques.

Lemma 81 *Every well-behaved process is adjacent to all processes of a minimal quorum.*

Proof. By Theorem 68, a well-behaved process p has at least one quorum q. Process p has an edge to every member of q. By Theorem 69, q is a superset of a minimal quorum q'. Therefore, p has an edge to every member of q'. ■

In Figure 4.3, process 4 is adjacent to all processes of $\{1, 2\}$.

Lemma 82 *Well-behaved processes of minimal quorums induce a strongly connected graph.*

Proof. Consider a pair of minimal quorums q_1 and q_2, and two well-behaved processes $p_1 \in q_1$ and $p_2 \in q_2$. The consistency property states that there is at least a well-behaved process p in the intersection of q_1 and q_2. By Theorem 80, the following edges are in the quorum graph: (p_1, p), (p, p_2), (p_2, p) and (p, p_1). Therefore, p_1 and p_2 are strongly connected. ■

In Figure 4.3, the processes $\{1, 2, 3\}$ are strongly connected.

We remember that the condensation of a graph is the graph resulted from contracting each of its strongly connected components to a single vertex. A condensation graph is a directed acyclic graph (DAG). DAGs have sink and source vertices. A component of the graph that is contracted to a sink vertex in the condensed graph is called a sink component.

Lemma 83 *All well-behaved processes in minimal quorums are in a sink component.*

Proof. By Theorem 82, the well-behaved processes of the minimal quorums are a strongly connected subgraph. Therefore, they fall in a component C. By Theorem 81, there are edges from the processes of every component to C. Therefore, C must be a sink component. ■

For example, in Figure 4.3, processes $\{1, 2, 3\}$ (shaded in green) are in the sink.

Lemma 84 *There exists a minimal quorum in every sink component.*

Proof. Consider a sink component S. By Theorem 81, there are edges from S to all processes of a minimal quorum q. This quorum q should be inside S. Otherwise, the fact that there are edges from S to q contradicts the assumption that S is a sink component. ■

Lemma 85 *Every quorum graph has only one sink component.*

Proof. The proof is by contradiction. If there are two sinks, by Theorem 84, each contains a minimal quorum. By the quorum intersection property, the two minimal quorums have an intersection; thus, the two sinks components intersect. However, components are disjoint. ■

Theorem 86 *All well-behaved processes of the minimal quorums are in the sink component.*

This is straightforward from Theorem 83 and Theorem 85. For example, in Figure 4.3, the well-behaved processes $\{1, 2, 3\}$ of the minimal quorums $\{1, 2\}$ and $\{1, 3, 5\}$ are in the sink.

Consider a reconfiguration from a quorum system $\mathcal{Q}$ to another $\mathcal{Q}'$, and a well-behaved process p. A *Leave* operation by p removes p from the set of active processes $\mathcal{A}$ *i.e.*, $p \notin \mathsf{dom}(\mathcal{Q}')$. Let q be an individual minimal quorum of p, *i.e.*, $q \in \mathcal{Q}(p)$. A *Remove*(q) operation by p removes q from the individual minimal quorum of p, *i.e.*, $q \notin \mathcal{Q}'(p)$.

Lemma 87 *Any leave or remove operation by a process outside the sink component of the quorum graph preserves consistency.*

Proof. By Theorem 72, it is sufficient to prove that quorum intersection is preserved for minimal quorums. By Theorem 86, the well-behaved intersections of minimal quorums fall in the sink component. Therefore, any leave or remove operation outside of the sink component preserves their quorum intersection. ■

Inspired by this result, our leave and remove protocols will avoid coordination when they are applied to a process that is outside of the sink component. (We will present a sink discovery protocol in the appendix section 4.13).

4.5 Reconfiguration and Trade-offs

In this section, we consider reconfigurations, how they can endanger the properties of a quorum system, and trade-off theorems for the properties that reconfiguration protocols can preserve. These trade-offs inform the design of our protocols in the next sections.

A process can request to *Join* or *Leave* the quorum system. It can further request to *Add* or *Remove* a quorum. However, a reconfiguration operation should not affect the safety and liveness of the quorum system.

Reconfiguration Attacks. Let $\mathcal{P} = \{1, 2, 3, 4\}$ where the Byzantine set is $\mathcal{B} = \{4\}$. Let the quorums of process 1 be $\mathcal{Q}(1) = \{\{1, 2, 4\}\}$. Similarly, let $\mathcal{Q}(2) = \{\{1, 2\}, \{2, 3\}\}$ and $\mathcal{Q}(3) = \{\{2, 3\}\}$. This quorum system enjoys quorum intersection for well-behaved processes since all pairs of quorums intersect at a well-behaved process. Let process 2 locally add a quorum $q_1 = \{2, 4\}$ its set of quorums $\mathcal{Q}(2)$. The quorum q_1 intersects all the existing quorums at the well-behaved process 2. Similarly, let process 3 locally add a quorum $q_2 = \{1, 3\}$ into its set of quorums $\mathcal{Q}(3)$. The quorum q_2 intersects all the existing

quorums at the well-behaved processes 1 or 3. Both reconfiguration requests seem safe, and if they are requested concurrently, they may be both permitted. However, the two new quorums do not intersect. An attacker can issue a transaction to spend some credit at process 2 with q_1, and another transaction to spend the same credit at process 3 with q_2. That leads to a double-spending and a fork. Even if processes 2 and 3 send their updated quorums to other processes, the attack can be successful if the time to send and receive updates is longer than the time to process a transaction. Similarly, a leave operation can lead to double-spending. In our example, if process 2 leaves the system, quorum intersection is lost. The reconfiguration protocols should preserve quorum intersection.

Trade-offs. We first formalize a few notions to state the trade-offs.

Reconfigurations. A reconfiguration changes a quorum system to another. We remember that a quorum system is a mapping from active well-behaved processes $\mathcal{A} \cap \mathcal{W}$ to their quorums. Consider a reconfiguration by a well-behaved process p that updates $\mathcal{Q}$ to $\mathcal{Q}'$. We consider four reconfiguration operations. The reconfiguration applies a *Join* operation by p iff $p \notin \mathsf{dom}(\mathcal{Q})$ and $p \in \mathsf{dom}(\mathcal{Q}')$ (*i.e.*, p is added to the active set $\mathcal{A}$). It applies an $Add(q)$ operation by p iff $q \in \mathcal{Q}'(p)$. It applies a *Leave* operation by p iff $p \in \mathsf{dom}(\mathcal{Q})$ and $p \notin \mathsf{dom}(\mathcal{Q}')$ (*i.e.*, p is removed from the active set $\mathcal{A}$). It applies a $Remove(q)$ operation by p where $q \in \mathcal{Q}(p)$ (*i.e.*, q is an individual minimal quorum of p) iff $q \notin \mathcal{Q}'(p)$.

Terminating. A reconfiguration protocol is terminating iff every operation by a well-behaved process eventually completes.

Each process declares its trust policy as its individual minimal quorums. A quorum should appear in the individual minimal quorums of a process only if that process has

$\mathcal{Q}_1(1) = _$ $\quad$ $\mathcal{Q}_2(1) = _$
$\mathcal{Q}_1(2) = \{\{2, 3\}, \{1, 2, 4\}\}$ $\quad$ $\mathcal{Q}_2(2) = \{\{2, 3\}, \{1, 2\}\}$
$\mathcal{Q}_1(3) = \{\{2, 3\}, \{1, 3, 4\}\}$ $\quad$ $\mathcal{Q}_2(3) = \{\{2, 3\}, \{3, 4\}\}$
$\mathcal{Q}_1(4) = \{\{1, 3, 4\}\}$ $\quad$ $\mathcal{Q}_2(4) = \{\{1, 3, 4\}\}$

Figure 4.4: Example Quorum Systems for Trade-offs

explicitly declared it as its quorum, during either the initialization or an add reconfiguration.

Policy-preservation. A *Leave* or *Remove* operation is policy-preserving iff it only removes individual minimal quorums. A *Join* operation is policy-preserving iff it does not change existing individual minimal quorums. An $Add(q)$ operation by a process p is policy-preserving iff it only adds q to individual minimal quorums of p.

Consistency-preservation. A reconfiguration is consistency-preserving iff it transforms a consistent quorum system to only a consistent one.

Availability-preservation. A reconfiguration is availability-preserving iff it only affects the availability of a process that is requesting *Leave*, or requesting *Remove* for its last quorum.

Theorem 88 *There is no* Leave *or* Remove *reconfiguration protocol that is policy-preserving, availability-preserving and terminating.*

Proof. The proof is by contradiction. We consider the Leave and Remove protocols in turn.

The Leave protocol: Consider the quorum system $\mathcal{Q}_1$ in Figure 4.4. Process 1 is Byzantine. $\mathcal{Q}_1(2) = \{\{2, 3\}\}$. (We will later reuse this example for the Remove protocol after adding the quorum $\{1, 2, 4\}$ for process 2, as the figure shows in color.) $\mathcal{Q}_1$ is available for $\{2, 3\}$. Process 2 requests to leave. Since the protocol is terminating, 2 eventually leaves,

and the quorum system is updated to $\mathcal{Q}'_1$. The quorum $\{2, 3\}$ makes 3 available in $\mathcal{Q}_1$ but not $\mathcal{Q}'_1$. If the protocol leaves the quorum $\{2, 3\}$ unchanged, then it includes the inactive process 2. The other quorum $\{1, 3, 4\}$ of 3 includes the Byzantine process 1. Thus, $\mathcal{Q}'_1$ does not preserve availability for 3. If the protocol removes 2 from the quorum $\{2, 3\}$, then $\mathcal{Q}'_1$ preserves availability for 3 but does not preserve policies.

The Remove protocol: We reuse the example above with a small change: $\mathcal{Q}_1(2) = \{\{2, 3\}, \{1, 2, 4\}\}$. Let process 2 remove quorum $\{2, 3\}$ and result in quorum system $\mathcal{Q}'_1$. Now, $\mathcal{Q}'_1$ loses availability for 2 unless process 2 removes 1 from its quorum $\{1, 2, 4\}$. However, that violates policies. ■

Theorem 89 *There is no* Add *reconfiguration protocol that is policy-preserving, consistency-preserving, and terminating.*

Proof. Consider the quorum system $\mathcal{Q}_2$ in Figure 4.4. Process 1 is Byzantine. Process 2 requests to add a new quorum $\{1, 2\}$ that is shown in color. Since the protocol is terminating, it will eventually add $\{1, 2\}$ to the quorums of 2, and result in the updated quorum system $\mathcal{Q}'_2$. The quorum system $\mathcal{Q}_2$ is consistent for $\{2, 3, 4\}$. However, in $\mathcal{Q}'_2$, the quorum $\{1, 2\}$ of process 2, and the quorum $\{1, 3, 4\}$ of the process 4 intersect at only the Byzantine process 1. Therefore, to preserve consistency, there are two cases. In the first case, $\{1, 3, 4\}$ is removed from the quorums of 4. Then, the well-behaved active process 4 has no quorums which violates the definition of heterogeneous quorum systems. In the second case, 2 is added to $\{1, 3, 4\}$. However, this violates policies for 4. ■

Algorithm 9: AC Leave and Remove (1/2)

```
1  Implements: Leave and Remove
2     request : Leave | Remove(q)
3     response : LeaveComplete | LeaveFail
4     RemoveComplete | RemoveFail
5  Variables:
6     Q
7     tomb : 2^P ← ∅
8     in-sink : Boolean, F : 2^P ← Discovery(Q)
9  Uses:
10    tob : TotalOrderBroadcast
11    apl : (∪Q) ∪ F ↦ AuthPPoint2PointLink
12 upon request Leave
13    if in-sink then
14       if ∀q1, q2 ∈ Q, (q1 ∩ q2)\{self} is self-blocking then
15          tob request Check(self, Q)
16       else
17          response LeaveFail
18    else
19       response LeaveComplete
20       apl(p) request Left(self)  for each p ∈ F
```

Algorithm 10: AC Leave and Remove (2/2)

```
1  upon response tob, Check(p', Q')
2     if ∃q1, q2 ∈ Q'. (q1 ∩ q2) \ ({p'} ∪ tomb) is not p'-blocking then
3        if p' = self then
4           response LeaveFail
5     else
6        tomb ← tomb ∪ {p'}
7        if p' = self then
8           response LeaveComplete
9           apl(p) request Left(self)  for each p ∈ F
10 upon response apl(p), Left(p)
11    Q ← {q \ {p} | q ∈ Q}
12 upon request Remove(q)
   ▷Handlers for Remove are similar to Leave except: The quorum q that should be removed is
     passed in the Check message (instead of Q) at line line 15, and the handler Check at line 1
     takes q as a parameter (instead of Q'). The update Q ← Q \ {q} is added after line 8.
```

4.6 Leave and Remove

In the light of these trade-offs, we next consider reconfiguration protocols. The protocols reconfigure an outlived quorum system into another. They assume that the given quorum system is outlived, *i.e.*, it has an outlived set of processes $\mathcal{O}$. In particular, they only require quorum inclusion (and not quorum sharing) inside $\mathcal{O}$. Let's now consider Leave and Remove protocols. (The Join protocol is straightforward and presented in section 4.9.) The client can issue a *Leave* request to leave the quorum system, and in return receives either a *LeaveComplete* or *LeaveFail* response. It can also issue the *Remove*(q) request to remove its quorum q, and in return receives either a *RemoveComplete* or *RemoveFail* response. Based on the trade-offs that we saw in Theorem 88, we present the availability-preserving and consistency-preserving protocols (AC protocols) in this section, and the policy-preserving and consistency-preserving protocols (PC protocols) in the appendix section 4.12.

Leave Protocol. We first consider the Leave protocol presented in Alg. 9, and then intuitively explain how it preserves the properties of the quorum system.

Variables and sub-protocols. Each process keeps its own set of individual minimal quorums Q. It also keeps the set *tomb* that records the processes that might have left. We saw that Theorem 87 presented an optimization opportunity for the coordination needed to preserve consistency: when the quorum system has quorum sharing, only processes in the sink component need coordination.

Therefore, each process stores whether it is in the sink component as the *in-sink* boolean, and its follower processes (*i.e.*, processes that have this process in their quorums) as the set F. (Processes can use a sink discovery protocol such as the one we present in

the appendix section 4.13. The sink information is just used for an optimization, and the protocol can execute without it.)

The protocol uses a total-order broadcast *tob*, and authenticated point-to-point links *apl* (to processes in the quorums Q and followers F). Total-order broadcast provides a broadcast interface on top of consensus [334, 311, 187, 289]. The consensus and total-order broadcast abstractions [289] require quorum intersection for safety, and quorum availability and inclusion for liveness. As we will show, the reconfiguration protocols preserve both of these properties for outlived quorum systems. The total-order broadcast ensures the following safety property: for every pair of messages m and m', and well-behaved processes p and p', if m is delivered before m' at p, then at p', the message m' is either not delivered or delivered after m. Further, it ensures the following liveness property: every outlived process will eventually deliver every message that a well-behaved process sends. We note that if a protocol naively uses *tob* to globally order and process reconfigurations, then since each process only knows its own quorums, it cannot independently check if the properties of the quorum system are preserved.

Protocol. When a process requests to leave (at line 12), it first checks whether it is in the sink component (at line 13). If it is not in the sink, then by Theorem 87, it can apply the optimizations that are shown with the blue color. The process can simply leave without synchronization (at line 19); it only needs to inform its follower set so that they can preserve their quorum availability. It sends a *Left* message to its followers (at line 20). Every well-behaved process that receives the message (at line 10) removes the sender from its quorums (at line 11). If the quorum system does not have quorum sharing or the sink

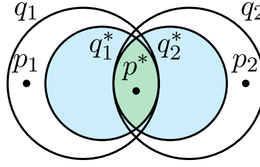


Figure 4.5: The Leave and Remove Protocols, Preserving Quorum Intersection.

information is not available, the protocol can be conservative (remove the blue lines) and always perform the coordination that we will consider next.

On the other hand, when the requesting process is in the sink component, its absence can put quorum intersection in danger. Therefore, it first locally checks a condition (at line 14). The check is just an optimization not to attempt leave requests that are locally known to fail. We will consider this condition in the next subsection. If the check fails, the leave request fails (at line 17). If the local check passes, the process broadcasts a *Check* request together with its quorums (at line 15). If processes receive and check concurrent leave requests in different orders, they may concurrently approve leave requests for all processes in a quorum intersection. Therefore, a total-order broadcast *tob* is used to enforce a total order for processing of *Check* messages. When a process receives a *Check* request with a set of quorums Q, it locally checks a condition for Q (at line 2). This check is similar to the check above but is repeated in the total order of deliveries by the *tob*. If the condition fails, the leave request fails (at line 4). If it passes, the leaving process is added to the *tomb* set (at line 6), and the leaving process informs its followers, and leaves (at lines 8 and 9).

Intuition. Let's now consider the checked condition and see how it preserves quorum intersection and inclusion.

Quorum Intersection. Let us first see an intuitive explanation of the condition, and why it preserves quorum intersection. We assume that the quorum system is outlived: there is a set of processes $\mathcal{O}$ such that the quorum system has quorum intersection at $\mathcal{O}$, quorum inclusion for $\mathcal{O}$, and quorum availability inside $\mathcal{O}$. As shown in Figure 4.5, consider well-behaved processes p_1 and p_2 with quorums q_1 and q_2 respectively, and let p^* be a process at the intersection of q_1 and q_2 in $\mathcal{O}$. The goal is to allow p^* to leave only if the intersection of q_1 and q_2 contains another process in $\mathcal{O}$. By the quorum inclusion property, p^* should have quorums q_1^* and q_2^* such that their well-behaved processes are included inside q_1 and q_2 respectively. Each process adds to its *tomb* set every process whose *Check* request passes. The total-order-broadcast *tob* delivers the *Check* requests in the same order across processes. Therefore, the result of the check and the updated *tomb* set is the same across processes after processing each request. Consider a *Check* request of a process p' which is ordered before that of p^*. If the check for p' is passed and it leaves, then the *tomb* set of p^* contains p'. Consider when the *Check* request of p^* is processed. The check ensures that p^* is approved to leave only if the intersection of q_1^* and q_2^* modulo the *tomb* set and p^* is p^*-blocking. By Theorem 75, since the quorum system is available inside $\mathcal{O}$, this means that the intersection of q_1^* and q_2^* after both p' and p^* leave still intersects $\mathcal{O}$. A process p in $\mathcal{O}$ remains in the intersection of q_1^* and q_2^*. Therefore, by quorum inclusion, p remains in the intersection of q_1 and q_2. Thus, outlived quorum intersection is preserved for q_1 and q_2.

Once the *tob* delivers the *Check* message of the leaving process p^* to p^* itself, it can locally decide whether it is safe to leave. We note that the local check ensures a global property: quorum intersection for the whole quorum system. We also note that

both quorum inclusion and quorum availability are needed to preserve quorum intersection. Further, we note that outlived quorum intersection is not affected if a Byzantine process leaves: the outlived processes where quorums intersect are by definition a subset of well-behaved processes.

Quorum inclusion. Now let us elaborate on the quorum inclusion property that we just used. When a process p' leaves, it sends *Left* messages to its followers (at either line 20 or line 9). The followers later remove p' from their quorums (at line 10-line 11). These updates are not atomic and happen over time. Therefore, there might be a window when a process p' is removed from the quorum q_1 (that we saw above), but not yet removed from q_1^*. Therefore, quorum inclusion only eventually holds. However, we observe that in the meanwhile, a weaker notion of quorum inclusion, that we call *active quorum inclusion*, is preserved. It considers inclusion only for the active set of processes $\mathcal{A} = \mathcal{P} \setminus \mathcal{L}$, *i.e.*, it excludes the subset $\mathcal{L}$ of processes that have already left. It requires the quorum q_1^* to be a subset of q_1 modulo $\mathcal{L}$. More precisely, it requires $q_1^* \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1$. This weaker notion is enough to preserve quorum intersection. In the above discussion for quorum intersection, the process p that remains in the intersection is not in the *tomb* set; therefore, it is an active process. Since it is in q_1^* and q_2^*, by active quorum inclusion, it will be in q_1 and q_2 as well.

Remove Protocol. Let us now consider the Remove protocol. Removing a quorum can endanger all the three properties of the quorum system: inclusion, availability, and even intersection. Consider a process p that removes a quorum q. (1) Let p be an outlived process, and let p' be a well-behaved process with quorum q' that includes p and q, but no other quorum of p. The removal of q, violates outlived quorum inclusion for q'. (2)

If q is the only quorum of p in the outlived set, the removal of q violates outlived availability for p. (3) As we saw above, p can lose outlived availability, and fall out of the outlived set. Consider a pair of quorums whose intersection includes only p from the outlived set. The removal of q violates outlived quorum intersection for these pair of quorums. Therefore, similar to a leaving process, a process that removes a quorum should coordinate, check the safety of its reconfiguration, and update others' quorums. As shown in Alg. 9, the Remove protocol is, thus, similar to the Leave protocol. The difference is that when a request to remove a quorum q is successful, q is removed from the quorums of the requesting process (after line 8).

Correctness. The Leave and Remove protocols maintain the properties of the quorum system. We prove that they preserve quorum intersection, and eventually provide quorum availability and quorum inclusion.

Let $\mathcal{L}$ denote the set of processes that receive a *LeaveComplete* or *RemoveComplete* response. As we saw before, processes in $\mathcal{L}$ may fall out of the outlived set. Starting from a quorum system that is outlived for $\mathcal{O}$, the protocols only eventually result in an outlived quorum system for $\mathcal{O} \setminus \mathcal{L}$. However, they preserve strong enough notions of quorum inclusion and availability, called active quorum inclusion and active quorum availability, which support quorum intersection to be constantly preserved. Consider a quorum q of p, and a process p' of q that falls in $\mathcal{L}$. Intuitively, active quorum inclusion for q does not require the inclusion of a quorum of p' in q, and active availability for p does not require p' to be well-behaved. We first capture these weaker notions and prove that they are preserved, and further, prove that quorum inclusion and availability eventually hold. We then use the

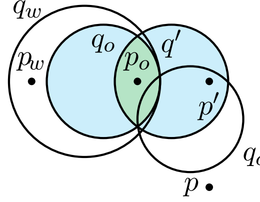


Figure 4.6: The Add protocol, Preserving Quorum Intersection.

above two preserved properties to prove that the protocols preserve quorum intersection at $\mathcal{O} \setminus \mathcal{L}$. The correctness theorems and proofs are available in section section 4.10 and section 4.14.

4.7 Add

We saw the trade-off for the add operation in Theorem 89. Since we never sacrifice consistency, we present an Add protocol that preserves consistency and availability. For brevity, we present an intuition and summary of the protocol in this section.

Example. Let us first see how adding a quorum for a process can violate the quorum inclusion and quorum intersection properties. Consider our running example from Figure 4.1. As we saw before, the outlive set is $\mathcal{O} = \{2, 3, 5\}$. If 3 adds a new quorum $\{3, 5\}$ to its set of quorums, it violates quorum inclusion for $\mathcal{O}$. The new quorum includes process 5 that is outlive. However, process 5 has only one quorum $\{2, 5\}$ that is not a subset of $\{3, 5\}$. Further, quorum intersection is violated since the quorum $\{1, 2, 4\}$ of 1 does not have a well-behaved intersection with $\{3, 5\}$.

Consider a quorum system that is outlive for a set of well-behaved processes $\mathcal{O}$, and a well-behaved process p that wants to add a new quorum q_n. (If the requesting process

p is Byzantine, it can trivially add any quorum. Further, we consider new quorums q_n that have at least one well-behaved process. Otherwise, no operation by the quorum is credible. For example, p itself can be a member of q_n.)

Intuition. Now, we explain the intuition of how the quorum inclusion and quorum intersection properties are preserved, and then an overview of the protocol.

Quorum inclusion. In order to preserve quorum inclusion, process p first asks each process in q_n whether it already has a quorum that is included in q_n. It gathers the processes that respond negatively in a set q_c. (In this overview, we consider the main case where there is at least one well-behaved process in q_c. The other cases are straightforward, and discussed in the proof of Theorem 103.) To ensure quorum inclusion, the protocol adds q_c as a quorum to every process p' in q_c. Since these additions do not happen atomically, quorum inclusion is only eventually restored. In order to preserve a weak notion of quorum inclusion called tentative quorum inclusion, each process stores a *tentative* set of quorums, in addition to its set of quorums. The protocol performs the following actions in order. It first adds q_c to the *tentative* quorums of every process in q_c, then adds q_n to the quorums of p, then adds q_c to the quorums of every process in q_c, and finally garbage-collects q_c from the *tentative* sets. Thus, the protocol preserves tentative quorum inclusion: for every quorum q, and outlive process p in q, there is either a quorum *or a tentative quorum* q' of p such that well-behaved processes of q' are included in q. We will see that when a process performs safety checks, it considers its tentative quorums in addition to its quorums.

Quorum Intersection. Existing quorums in the system have outlive quorum intersection, *i.e.*, quorum intersection at $\mathcal{O}$. We saw that when a process p wants to add

a new quorum q_n, a quorum $q_c \subseteq q_n$ may be added as well. We need to ensure that q_c is added only if outlive quorum intersection is preserved. We first present the design intuition, and then an overview of the steps of the protocol.

The goal is to approve adding q_c as a quorum only if it has an outlive intersection with every other quorum q_w in the system. Theorem 75 presents an interesting opportunity to check this condition locally. It states that if an outlive process finds a set self-blocking, then that set has an outlive process. Thus, if we can pass the quorum q_c to an outlive process p_o, and have it check that $q_c \cap q_w$ is self-blocking, then we have that the intersection of q_c and q_w has an outlive process. However, no outlive process is aware of all quorums q_w in the system. Tentative quorum inclusion can help here. If the outlive process p_o is inside p_w, then by outlive quorum inclusion, p_o has a quorum or tentative quorum q_o (whose well-behaved processes are) included in p_w. The involved quorums are illustrated in Figure 4.7. If the outlive process p_o check for all its own quorums or tentative quorums q (including q_o), that $q_c \cap q$ is self-blocking then, by Theorem 75, the intersection of q_c and q_o has an outlive process. Since q_o is included in q_w, we get the desire result that the intersection of q_c and q_w has an outlive process. However, how do we reach from the requesting process p to an outlive process p_o in every quorum q_w in the system? Process p that is requesting to add the quorum q_c doesn't know whether it is outlive itself. There is at least a well-behaved process p' in the quorum q_c. By outlive quorum intersection, every quorum q' of p' intersects with every other quorum q_w at an outlive process p_o. Therefore, the protocol takes two hops to reach to the outlive process p_o: process p asks the processes p' of q_c, and then p' asks the processes p_o in its quorums q'.

Based on the intuition above, we now consider an overview of the protocol. Before adding q_c as a quorum for each process in q_c, the protocol goes through two hops. In the first hop, process p asks each process p' in q_c to perform a check. In the second hop, process p' asks its quorums q' to perform a check. A process p_o in q' checks for every quorum q_o in its set of quorums and tentative quorums that $q_o \cap q_c$ is self-blocking. If the check passes, p_o sends an Ack to p'; otherwise, it sends a Nack. Process p' waits for an Ack from at least one of its quorums q' before sending a commit message back to p. Once p receives a commit message from each process in q_c, it safely adds q_n to its own set of quorums, and requests each process in q_c to add q_c as a quorum.

In the limited space, we presented an overview of the add protocol. The details of the protocol and its correctness proofs are available in the appendix section 4.11 and section 4.15.

4.8 Sink Discovery

Following the graph characterization that we saw in section 4.4, we now present a decentralized protocol that can find whether each process is in the sink component of the quorum graph. We first describe the protocol and then its properties.

Protocol. Consider a quorum system with quorum intersection, availability and sharing. The sink discovery protocol in Alg. 17 finds whether each well-behaved process is in the sink. It also finds the set of its followers. A process p is a follower of process p' iff p has a quorum that includes p'. The protocol has two phases. In the first, it finds the well-behaved minimal quorums, *i.e.*, every minimal quorum that is a subset of well-

behaved processes. Since well-behaved minimal quorums are inside the sink, the second phase extends the discovery to other processes in the same strongly connected component.

Variables and sub-protocols. In the quorum system $\mathcal{Q}$, each process **self** stores its own set of individual minimal quorums $Q = \mathcal{Q}(\mathbf{self})$, a map *qmap* from other processes to their quorums which is populated as processes communicate, the *in-sink* boolean that stores whether the process is in the sink, and the set of follower processes F. The protocol uses authenticated point-to-point links *apl*. They provide the following safety and liveness properties. If the sender and receiver are both well-behaved, then the message will be eventually delivered. Every message that a well-behaved process delivers from a well-behaved process is sent by the later, *i.e.*, the identity of a well-behaved process cannot be forged.

Protocol. When processes receive a *Discover* request (at line 9), they exchange their quorums with each other. In this first phase, each process sends an *Exchange* message with its quorums Q to all processes in its quorums. When a process receives an *Exchange* message (at line 11), it adds the sender to the follower set F, and stores the received quorums in its *qmap*. As *qmap* is populated, when a process finds that one of its quorums q is a quorum of every other process in q as well (at line 14), by Theorem 79, it finds that its quorum q is a minimal quorum, and by Theorem 86, it finds itself in the sink. Thus, it sets its *in-sink* variable to `true` in the first phase (at line 15).

The process then sends an *Extend* message with the quorum q to all processes of its own quorums Q (at line 16). The *Extend* messages are processed in the second phase. The processes of every well-behaved minimal quorum are found in this phase. In Figure 4.3, since the quorum $\{1, 2\}$ is a quorum for both of 1 and 2, they find themselves in the sink.

However, process 3 might receive misleading quorums from process 5, and hence, may not find itself in the sink in this phase.

The processes P_1 of every well-behaved minimal quorum find themselves to be in the sink in the first phase. Let P_2 be the well-behaved processes of the remaining minimal quorums. A pair of minimal quorums have at least a well-behaved process in their intersection. In Figure 4.3, the two minimal quorums $P_1 = \{1, 2\}$ and $P_2 = \{1, 3, 5\}$ intersect at 1. Therefore, by Theorem 80, every process in P_2 is a neighbor of a process in P_1. Thus, in the second phase, the processes P_1 can send *Extend* messages to processes in P_2, and inform them that they are in the sink. In Figure 4.3, process 1 can inform process 3. The protocol lets a process accept an *Extend* message containing a quorum q only when the same message comes from the intersection of q and one of its own quorums q' (at line 17). Let us see why a process in P_2 cannot accept an *Extend* message from a single process. A minimal quorum q that is found in phase 1 can have a Byzantine process p_1. Process p_1 can send an *Extend*(q) message (even with signatures from members of q) to a process p_2 even if p_2 is not a neighbor of p_1, and make p_2 believe that it is in the sink. In Figure 4.3, the Byzantine process 5 can collect the quorum $\{1, 3, 5\}$ from 1 and 3, and then send an *Extend* message to 4 to make 4 believe that it is inside the sink. Therefore, a process p_2 in P_2 accepts an *Extend*(q) message only when it is received from the intersection of q and one of its own quorums. Since there is a well-behaved process in the intersection of the two quorums, process p_2 can then trust the *Extend* message. When the check passes, p_2 finds itself to be in the sink, and sets the *in-sink* variable to `true` in the second phase (at line 18). In Figure 4.3, when process 3 receives an *Extend* message with quorum $\{1, 2\}$ from 1, since

$\{1\}$ is the intersection of the quorum $\{1, 3, 5\}$ of 3, and the received quorum $\{1, 2\}$, process 3 accepts the message.

Algorithm 11: Sink Discovery Protocol

1 **Variables:**
2 Q ▷ The individual minimal quorums of **self**
3
4 $qmap : \mathcal{P} \mapsto \mathsf{Set}[2^{\mathcal{P}}]$
5 $in\text{-}sink : \mathsf{Boolean} \leftarrow \mathsf{false}$
6 $F : 2^{\mathcal{P}}$
7 **Uses:**
8 $apl : \mathcal{P} \mapsto \mathsf{AuthPerfectPointToPointLink}$
9 **upon request** *Discover*
10 $apl(p)$ **request** $Exchange(Q)$ for each $p \in \cup Q$
11 **upon response** $apl(p)$, $Exchange(Q')$
12 $F \leftarrow F \cup \{p\}$
13 $qmap(p) \leftarrow Q'$
14 **upon** $\exists q \in Q.\ \forall p \in q.\ q \in qmap(p)$
15 $in\text{-}sink \leftarrow \mathsf{true}$
16 $apl(p)$ **request** $Extend(q)$ for each $p \in \cup Q$
17 **upon response** $\overline{apl(p)}, Extend(q)$ s.t. $\exists q' \in Q.\ \{\overline{p}\} = q \cap q'$
18 $in\text{-}sink \leftarrow \mathsf{true}$

Let *ProtoSink* denote the set of well-behaved processes where the protocol sets the *in-sink* variable to true. The discovery protocol is complete: all the well-behaved processes of minimal quorums will eventually know that they are in the sink (*i.e.*, set their *in-sink* to true).

Lemma 90 (Completeness) *For all $q \in MQ(\mathcal{Q})$, eventually $q \cap \mathcal{W} \subseteq ProtoSink$.*

This result brings an optimization opportunity: the leave and remove protocols can coordinate only when a process inside *ProtoSink* is updated. Although, completeness is sufficient for safety of the optimizations, in the appendix section 4.16, we prove both the

completeness and accuracy of the two phases in turn. The accuracy property states that *ProtoSink* is a subset of the sink component.

4.9 Join

Algorithm 12: Join

```
1  Implements: Join
2    request : Join(ps)
3    response : JoinComplete
4  Variables:
5    Q : Set[2^P] ▷ Individual minimal quorums of self
6
7    S : Set[2^P]
8    F : Set[P]▷ Followers
9
10   qmap : P ↦ Set[2^P]
11 Uses:
12   apl : P ↦ AuthPerfectPointToPointLink
13 upon request Join(ps)
14   S ← {{q}}, qmap ← ∅
15 upon ∃q ∈ S, ∃p ∈ q, s.t. qmap(p) = ∅
16   apl(p) request Prob
17 upon response apl(p'), Prob
18   F ← F ∪ {p'}
19   apl(p') request Quorums(Q)
20 upon response apl(p') Quorums(Q')
21   qmap ← qmap[p' ↦ Q']
22   for each q ∈ S s.t. p' ∈ q
23     for each q' ∈ Q'
24       S ← S \ {q} ∪ {q ∪ q'}
25 upon ∀q ∈ S. ∀p ∈ q. ∃q' ∈ qmap(p). q' ⊆ q
26   Q ← S
27   response JoinComplete
```

We now consider the Join protocol which is presented in Alg. 12. When a process p wants to join the system, it issues a *Join* request with an initial set of processes ps, which is a set of processes that it trusts (at line 13). In order to maintain the quorum inclusion property, the requesting process starts with ps as a tentative quorum and probes these

processes for their quorums (at line 15-line 16). When a process receives a probe request, it sends back it's quorums, and adds the sender to its follower set (at line 17-line 19). When a quorum from a process p is received, it is added to each tentative quorum that contains p (at line 20-line 24). The tentative quorums grow and probing continues for the new members. It stops when the tentative quorums are quorum including (at line 25).

Correctness. We now show that the Join protocol preserves all the three properties of the quorum system.

Lemma 91 *For every quorum system and well-behaved set of processes $\mathcal{O}$, the* Join *protocol preserves quorum intersection at $\mathcal{O}$, quorum availability for $\mathcal{O}$, and quorum inclusion for $\mathcal{O}$. Therefore, it preserves every outlived set for the quorum system. Further, newly joined processes have quorum inclusion.*

Proof. Quorum intersection at $\mathcal{O}$ is preserved since the existing quorums have intersection at $\mathcal{O}$, and the new quorums are supersets of existing quorums. Quorum availability and quorum inclusion for $\mathcal{O}$ are preserved since the quorums of existing processes do not change. Therefore, by the three properties above, it preserves every outlived set $\mathcal{O}$. Further, a newly joined process has quorum inclusion, since the new quorums pass the condition at line 25 which is sufficient for quorum inclusion. ■

We add that for adding a quorum, if adding a superset of the given quorum is considered policy-preserving, then the add protocol can be similar to the join protocol above.

4.10 AC Leave and Remove

4.10.1 Correctness

We now state that Leave and Remove protocols maintain the properties of the quorum system. We prove that they preserve quorum intersection, and eventually provide quorum availability and quorum inclusion.

Starting from an outlived quorum system, the protocols only eventually result in an outlived quorum system. However, they preserve strong enough notions of quorum inclusion and availability, called active quorum inclusion and active quorum availability, which support quorum intersection to be constantly preserved. We first capture these weaker notions and prove that they are preserved, and further, prove that quorum inclusion and availability eventually hold. We then use the above two preserved properties to prove that quorum intersection is preserved. All in all, we show that the protocols maintain quorum intersection as a safety property, and quorum availability and inclusion as liveness properties.

Let $\mathcal{L}$ denote the set of processes that have received a *LeaveComplete* or *RemoveComplete* response. As we saw before, processes in $\mathcal{L}$ may fall out of the outlived set.

Quorum inclusion. Active quorum inclusion captures inclusion modulo the set $\mathcal{L}$.

Definition 92 (Active quorum inclusion) *A quorum system $\mathcal{Q}$ has* active *quorum inclusion for P iff for all well-behaved processes p and quorums q of p, if a process p' in q is inside P, then there is a quorum q' of p' such that well-behaved and active processes of q' are a subset of q i.e., $\forall p \in \mathcal{W}.\ \forall q \in \mathcal{Q}(p).\ \forall p' \in q \cap P.\ \exists q' \in \mathcal{Q}(p').\ q' \cap \mathcal{W} \setminus \mathcal{L} \subseteq q$.*

It is obvious that quorum inclusion implies active quorum inclusion. We now state that active quorum inclusion is preserved, and quorum inclusion is eventually reconstructed.

Lemma 93 (Preservation of Quorum inclusion) *The AC Leave and Remove protocols preserve active quorum inclusion. Further, starting from a quorum system that has quorum inclusion for processes $\mathcal{O}$, the protocols eventually result in a quorum system with quorum inclusion for $\mathcal{O} \setminus \mathcal{L}$.*

Quorum Availability. Let's define the notions of active availability and active blocking sets.

Definition 94 (Active Availability) *A quorum system has active availability inside a set of processes P iff every process p in $P \setminus \mathcal{L}$ has at least a quorum q such that $q \setminus \mathcal{L}$ is in P.*

It is obvious that availability implies active availability.

Definition 95 (Active Blocking Set) *A set of processes P is an active blocking set for a process p iff for every quorum q of p, the set $q \setminus \mathcal{L}$ intersects P.*

Lemma 96 *For every quorum system that has active availability inside P, every active blocking set of every process in P intersects $P \setminus \mathcal{L}$.*

The proof is similar to the proof of Theorem 75. We use this lemma to show that active availability is preserved, and availability is eventually reconstructed.

Lemma 97 (Preservation of Availability) *The AC Leave and Remove protocols preserve active availability. Further, starting from a quorum system that has availability inside processes $\mathcal{O}$, the protocols eventually result in a quorum system with availability inside $\mathcal{O} \setminus \mathcal{L}$.*

Quorum Intersection. We saw that the protocols preserve active availability and active quorum inclusion. We use these two properties to show that they preserve quorum intersection.

Lemma 98 (Preservation of Quorum Intersection) *If a quorum system has quorum intersection at processes $\mathcal{O}$, active availability inside $\mathcal{O}$, and active quorum inclusion for $\mathcal{O}$, then the AC* Leave *and* Remove *protocols preserve quorum intersection at $\mathcal{O} \setminus \mathcal{L}$.*

Outlive. The three lemmas that we saw show that an outlived quorum system is eventually reconstructed.

Lemma 99 (Preservation of Outlived set) *Starting from a quorum system that is outlived for processes $\mathcal{O}$, the AC* Leave *and* Remove *protocols eventually result in a quorum system that is outlived for $\mathcal{O} \setminus \mathcal{L}$.*

Immediate from Theorem 98, Theorem 93, and Theorem 97.

The proofs are available in the appendix section 4.14.

4.11 Add

We saw the trade-off for the add operation in Theorem 89. Since we never sacrifice consistency, we present an Add protocol that preserves consistency and availability.

Example. Let us first see how adding a quorum for a process can violate the quorum inclusion and quorum intersection properties. Consider our running example from Figure 4.1. As we saw before, the outlived set is $\mathcal{O} = \{2, 3, 5\}$. If 3 adds a new quorum $\{3, 5\}$ to its set of quorums, it violates quorum inclusion for $\mathcal{O}$. The new quorum includes

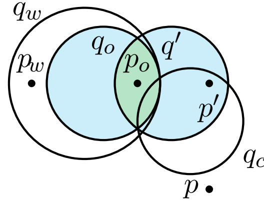


Figure 4.7: The Add protocol, Preserving Quorum Intersection.

process 5 that is outlive. However, process 5 has only one quorum $\{2, 5\}$ that is not a subset of $\{3, 5\}$. Further, quorum intersection is violated since the quorum $\{1, 2, 4\}$ of 1 does not have a well-behaved intersection with $\{3, 5\}$.

Consider a quorum system that is outlived for a set of well-behaved processes $\mathcal{O}$, and a well-behaved process p that wants to add a new quorum q_n. (If the requesting process p is Byzantine, it can trivially add any quorum. Further, we consider new quorums q_n that have at least one well-behaved process. Otherwise, no operation by the quorum is credible. For example, p itself can be a member of q_n.)

Intuition. Now, we explain the intuition of how the quorum inclusion and quorum intersection properties are preserved, and then an overview of the protocol.

Quorum inclusion. In order to preserve quorum inclusion, process p first asks each process in q_n whether it already has a quorum that is included in q_n. It gathers the processes that respond negatively in a set q_c. (In this overview, we consider the main case where there is at least one well-behaved process in q_c. The other cases are straightforward, and discussed in the proof of Theorem 103.) To ensure quorum inclusion, the protocol adds q_c as a quorum to every process p' in q_c. Since these additions do not happen atomically, quorum inclusion is only eventually restored. In order to preserve a weak notion of quorum

inclusion called tentative quorum inclusion, each process stores a *tentative* set of quorums, in addition to its set of quorums. The protocol performs the following actions in order. It first adds q_c to the *tentative* quorums of every process in q_c, then adds q_n to the quorums of p, then adds q_c to the quorums of every process in q_c, and finally garbage-collects q_c from the *tentative* sets. Thus, the protocol preserves tentative quorum inclusion: for every quorum q, and outlived process p in q, there is either a quorum *or a tentative quorum* q' of p such that well-behaved processes of q' are included in q. We will see that when a process performs safety checks, it considers its tentative quorums in addition to its quorums.

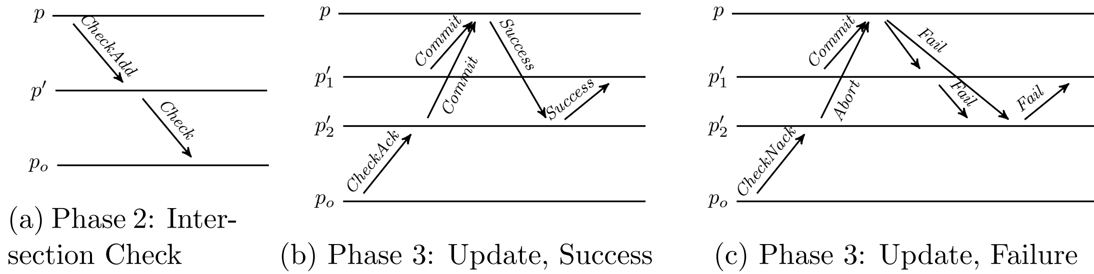
Quorum Intersection. Existing quorums in the system have outlived quorum intersection, *i.e.*, quorum intersection at $\mathcal{O}$. We saw that when a process p wants to add a new quorum q_n, a quorum $q_c \subseteq q_n$ may be added as well. We need to ensure that q_c is added only if outlived quorum intersection is preserved. We first present the design intuition, and then an overview of the steps of the protocol.

The goal is to approve adding q_c as a quorum only if it has an outlived intersection with every other quorum q_w in the system. Theorem 75 presents an interesting opportunity to check this condition locally. It states that if an outlived process finds a set self-blocking, then that set has an outlived process. Thus, if we can pass the quorum q_c to an outlived process p_o, and have it check that $q_c \cap q_w$ is self-blocking, then we have that the intersection of q_c and q_w has an outlived process. However, no outlived process is aware of all quorums q_w in the system. Tentative quorum inclusion can help here. If the outlived process p_o is inside p_w, then by outlived quorum inclusion, p_o has a quorum or tentative quorum q_o (whose well-behaved processes are) included in p_w. The involved quorums are illustrated in

Figure 4.7. If the outlived process p_o check for all its own quorums or tentative quorums q (including q_o), that $q_c \cap q$ is self-blocking then, by Theorem 75, the intersection of q_c and q_o has an outlived process. Since q_o is included in q_w, we get the desire result that the intersection of q_c and q_w has an outlived process. However, how do we reach from the requesting process p to an outlived process p_o in every quorum q_w in the system? Process p that is requesting to add the quorum q_c doesn't know whether it is outlived itself. There is at least a well-behaved process p' in the quorum q_c. By outlived quorum intersection, every quorum q' of p' intersects with every other quorum q_w at an outlived process p_o. Therefore, the protocol takes two hops to reach to the outlived process p_o: process p asks the processes p' of the quorum q_c, and then p' asks the processes p_o in its quorums q'.

Based on the intuition above, we now consider an overview of the protocol. Before adding q_c as a quorum for each process in q_c, the protocol goes through two hops. In the first hop, process p asks each process p' in q_c to perform a check. In the second hop, process p' asks its quorums q' to perform a check. A process p_o in q' checks for every quorum q_o in its set of quorums and tentative quorums that $q_o \cap q_c$ is self-blocking. If the check passes, p_o sends an Ack to p'; otherwise, it sends a Nack. Process p' waits for an Ack from at least one of its quorums q' before sending a commit message back to p. Once p receives a commit message from each process in q_c, it safely adds q_n to its own set of quorums, and requests each process in q_c to add q_c as a quorum.

Protocol Summary. We now present a summary of the protocol. (The details of the protocol are available in section 4.11). A process p issues an $Add(q_n)$ request in order to add the quorum q_n to its set of individual minimal quorums, and receives either



(a) Phase 2: Intersection Check (b) Phase 3: Update, Success (c) Phase 3: Update, Failure

Figure 4.8: Phase 2: Intersection Check, and Phase 3: Update

an *AddComplete* or *AddFail* response. The protocol has three phases: inclusion check, intersection check and update.

Phase 1: Inclusion Check. In phase 1, upon an $Add(q_n)$ request, the requesting process p first checks if quorum inclusion would be preserved for q_n. It sends out $Inclusion(q_n)$ messages to processes in q_n. When a process p' receives the message, it checks whether it already has a quorum which is a subset of q_n, and accordingly sends either *AckInclusion* or *NackInclusion*. Upon receiving these responses, the requesting process p adds the sender p' to the *ack* or *nack* sets respectively. The set *nack* is the set of processes that do not have quorum inclusion. Upon receiving acknowledgment from all processes in q_n, if *nack* is empty, then p simply adds q_n to its set of quorums before issuing the *AddComplete* response. Otherwise, the set $q_c = nack$ is the quorum that should be added to the set of quorums for each process in q_c. To make sure this addition preserves quorum intersection, process p starts phase 2 by sending $CheckAdd(q_c)$ to processes in q_c.

Phase 2: Intersection Check. In phase 2 (Figure 4.8a), when a $CheckAdd(q_c)$ request is received at a process p', it adds q_c to its tentative set, and sends out a *Check* message to all its quorums. When a process q_o delivers a *Check*, it checks that the intersection of the new quorum q_c with each of its own quorums and tentative quorums is self-blocking.

As we saw in the intuition part, this check ensures that there is an outlived process in the intersection. If the checks pass, p_o sends a *CheckAck* message back to p'. Otherwise, it sends a *CheckNack*.

Phase 3: Update. In phase 3 (Figure 4.8b and Figure 4.8c), once a process p' receives *CheckAck* messages from one of its quorums, it sends a *Commit* message to the requesting process p. On the other hand, if it receives *CheckNack* from one of its blocking sets, then there is no hope of receiving *CheckAck* from a quorum, and it sends an *Abort* message to p. The requesting process p succeeds if it receives a *Commit* message from every process in the quorum q_c. It fails if it receives an *Abort* message from at least one of them. On the success path, process p adds q_n to its own set of quorums, and sends a *Success* message to processes in q_c before issuing the *AddComplete* response. The *Success* message includes a signature from each process in q_c. On the failure path, process p sends a *Fail* message together with a signature to each process in q_c before issuing an *AddFail* response.

Attack scenarios. Let us consider attack scenarios that motivate the design decisions, and then we get back to the protocol. If the requesting process p is Byzantine, it may send a *Success* message to some processes in q_c, and a *Fail* message to others. Then, a process that receives *Success* adds q_c to its quorums, and another process that receives *Fail* removes q_c from its *tentative* quorums. This would break tentative quorum inclusion. To prevent this, every process p' that receives a *Fail* message echos it to other processes in q_c, and accepts a *Fail* message only when it has received an echo from every process in q_c. Further, a process that accepts a *Success* does not later accept a *Fail* and vice versa. Therefore, we will have the safety invariant that once a process accepts a *Fail*, no other

process accepts a *Success*, and vice versa. Another attack scenario is that the requesting process p just sends a *Success* message to some processes in q_c and not others. This attack does not break any of the properties; however, inhibits the progress of other processes. Therefore, every process p' that receives a *Success* message echos it to other processes in q_c.

Success. A process p' accepts a *Success* message to add q_c only if it has not already accepted a *Success* or *Fail* message, and the message comes with valid signatures from all processes in q_c. The signatures are needed to prevent receiving a fake *Success* message from a Byzantine process. Process p' first echos the *Success* message to other processes in q_c. It then adds q_c to its set of quorums, and removes q_c from its tentative quorums.

Failure. A process p' receives a *Fail* message for q_c only if it has not already accepted a *Success* or *Fail* message, and the message comes with a valid signature from the requesting process p. This signature prevents receiving fake *Fail* messages from Byzantine processes. Process p' echos the message, and adds the sender to a set. Once this sender set contains all the processes of q_c, it accepts the *Fail* message, and removes q_c from its tentative quorums.

4.11.1 Protocol

We present the protocol in three parts: Alg. 13, Alg. 14, and Alg. 15. A well-behaved process p issues an $Add(q_n)$ request in order to add the quorum q_n to its set of individual minimal quorums, and receives either an *AddComplete* or *AddFail* response.

Variables and sub-protocols. (1) Each process stores its own set of quorums Q, (2) two sets of processes *ack* and *nack*, (3) a set *tentative* that stores the set of tuples

$\langle p, q_c \rangle$ where q_c is a quorum that process p has asked to add, (4) a map *failed* that maps pairs $\langle p, q_c \rangle$ of the requesting process p and the quorum q_c that p wants to add, to the set of processes that a fail message is received from, and (5) a map *succeeded* from the same domain to boolean. The protocol uses a total-order broadcast *tob*, and authenticated point-to-point links *apl*.

The protocol is executed in three phases: inclusion check, intersection check and update.

Phase 1: Inclusion Check. In phase 1 (Alg. 13 and Figure 4.9), upon an $Add(q_n)$ request (at line 13), the requesting process p first checks if quorum inclusion would be preserved for q_n. It sends out $Inclusion(q_n)$ messages to processes in q_n (at line 14). When a process p' receives the message (at line 15), it checks whether it already has a quorum which is a subset of q_n (at line 16), and accordingly sends either *AckInclusion* or *NackInclusion* (at line 17 and line 19). Upon receiving these responses, the requesting process p adds the sender p' to the *ack* or *nack* sets respectively (at line 21 and line 23). The set *nack* is the set of processes that do not have quorum inclusion. Upon receiving acknowledgment from all processes in q_n (at line 24), if *nack* is empty (at line 25), then q_n is simply added to the set of quorums before issuing the *AddComplete* response. Otherwise, the set *nack* is

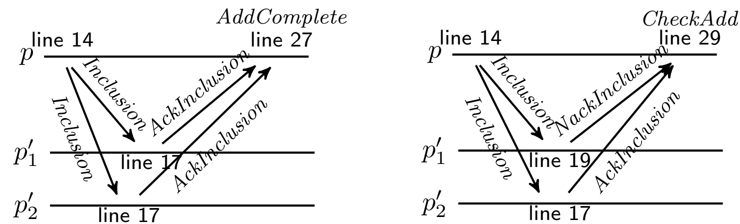


Figure 4.9: Phase1: Inclusion Check

Algorithm 13: Add quorum (Phase 1: Inclusion check)

```
1  Implements: Add
2    request : Add(q_n)
3    response : AddComplete | AddFail
4  Variables:
5    Q ▷ The individual minimal quorums of self
6
7    ack, nack : 2^P ← ∅
8    tentative : Set[P, 2^P] ← ∅
9    failed : ⟨P, 2^P⟩ ↦ 2^P ← ∅
10   succeeded : ⟨P, 2^P⟩ ↦ Boolean ← false (overlined)
11 Uses:
12   apl : (∪Q) ∪ q_n ↦ AuthPerfectPointToPointLink
13 upon request Add(q_n)
14   apl(p) request Inclusion(q_n) for each p ∈ q_n
15 upon response apl(p), Inclusion(q_n)
16   if ∃q ∈ Q. q ⊆ q_n then
17     apl(p) request AckInclusion
18   else
19     apl(p) request NackInclusion
20 upon response apl(p), AckInclusion
21   ack ← ack ∪ {p}
22 upon response apl(p), NackInclusion
23   nack ← nack ∪ {p}
24 upon ack ∪ nack = q_n
25   if nack = ∅ then
26     Q ← Q ∪ {q_n}
27     response AddComplete
28   else
29     apl(p′) request CheckAdd(nack) for each p′ ∈ nack
```

the quorum q_c that should be added to the set of quorums for each of its members. To make sure this addition preserves quorum intersection, process p starts phase 2 by sending *CheckAdd*(*nack*) to processes in *nack* (at line 29).

Algorithm 14: Add quorum (Phase 2: Intersection check)

```
29 upon response apl(p), CheckAdd(q_c)
30 |  tentative ← tentative ∪ ⟨p, q_c⟩
31 |_ apl(p_o) request Check(self, q_c) for each p_o ∈ ∪Q
32 upon response apl(p'), Check(p, p', q_c)
33 |  let overline(⟨_, q'_c⟩) := tentative in
34 |  if ∀q ∈ ∪{overline(q'_c)} ∪ Q. q_c ∩ q is self-blocking then
35 |  |_ apl(p') request CheckAck(p, q_c)
36 |  else
37 |_ |_ apl(p') request CheckNack(p, q_c)
```

Phase 2: Intersection Check. In phase 2 (Alg. 14 and Figure 4.8a), when a *CheckAdd*(q_c) request is received at a process p' (at line 29), it adds q_c to its tentative set (at line 30), and it sends out a *Check* message to all its quorums (at line 31). When a process q_o delivers a *Check* (at line 32), it checks that the intersection of the new quorum q_c with each of its own quorums in Q and its tentative quorums in *tentative* is **self**-blocking (at line 33-line 34). As we saw in the overview part of this section, this check ensures that there is an outlived process in the intersection. Then, the process p_o sends a *CheckAck* message back to p' (at line 35). Otherwise, it sends a *CheckNack* message (at line 37).

Phase 3: Update. In phase 3 (Alg. 15, Figure 4.8b and Figure 4.8c), once a process p' receives *CheckAck* messages from one of its quorums (at line 38), it sends a *Commit* message to the requesting process p (at line 39). On the other hand, if it receives *CheckNack* from one of its blocking sets (at line 40), then there is no hope of receiving *CheckAck* from a quorum, and it sends an *Abort* message to the requesting process p (at

line 41). The requesting process p succeeds if it receives a *Commit* message from every process in the quorum q_c. It fails if it receives an *Abort* message from at least one of them. On the success path (at line 42), process p adds q_n to its own set of quorums (at line 43), and sends a *Success* message to processes in q_c (at line 44) before issuing the *AddComplete* response. The *Success* message includes a signature from each process in q_c. On the failure path (at line 51), process p sends a *Fail* message together with a signature to each process in q_c before issuing an *AddFail* response.

Attack scenarios. Let us consider attack scenarios that motivate the design decisions, and then we get back to the protocol. If the requesting process p is Byzantine, it may send a *Success* message to some processes in q_c, and a *Fail* message to others. Then, a process that receives *Success* adds q_c to its quorums, and another process that receives *Fail* removes q_c from its *tentative* set. This would break tentative quorum inclusion. To prevent this, every process p' that receives a *Fail* message echos it to other processes in q_c, and processes a *Fail* message only when it has received its echo from every process in q_c. Further, a process that receives a *Success* does not later accept a *Fail*. Therefore, we will have the safety invariant that once a process accepts a *Fail*, no other process accepts a *Success*, and vice versa. Another attack scenario is that the requesting process p just sends a *Success* message to some processes in q_c and not others. This attack does not break any of the properties; however, inhibits the progress of other processes. Therefore, every process p' that receives a *Success* message echos it to other processes in q_c.

Success. A process p' accepts a *Success* message to add q_c (at line 46) only if the *succeeded* is not set (*i.e.*, p' has not already received a *Success* message, as an optimization),

and the message comes with valid signatures from all processes in q_c. The signatures are needed to prevent receiving a fake *Success* message from a Byzantine process. Process p' first echos the *Success* message to other processes in q_c (at line 48). It then adds q_c to its set of quorums (at line 49), sets *succeeded* to `true` (at line 47), and removes q_c from the *tentative* set (at line 50).

Failure. A process p' accepts a *Fail* message for q_c (at line 54) only if *succeeded* is not set (*i.e.*, p' has not received a *Success* message), and the message comes with a valid signature from the requesting process p. This signature prevents receiving fake *Fail* messages from Byzantine processes. Process p' echos the message when it comes from the requesting process p (at line 55-line 56), and adds the sender to the set $\mathit{failed}(p, q_c)$ (at line 57). Once this set contains all the processes of q_c, it removes q_c from its *tentative* set (at line 58).

4.11.2 Correctness

We now show that the `Add` protocol preserves consistency and availability. First we show that although it preserves quorum inclusion only eventually, it does preserve a weak notion of quorum inclusion. We later show that this notion is strong enough to preserve quorum intersection.

Quorum inclusion. In order to preserve quorum inclusion, when q_n is in the system, q_c should be in the system as well. The protocol adds the new quorum q_n for a process p only after q_c is added to the *tentative* set of each process p' in q_c. It then removes q_c from the *tentative* set of p' only after q_c is added to the set of quorums Q of p'. Therefore,

Algorithm 15: Add quorum (Phase 3: Update)

38 **upon response** $\overline{apl(p_o), CheckAck(p, q_c)}$ s.t. $\{\overline{p_o}\} \in Q$
39 $\quad$ $apl(p)$ **request** $Commit(q_c)^{sig}$
40 **upon response** $\overline{apl(p_o), CheckNack(p, q_c)}$ s.t. $\{\overline{p_o}\}$ is **self**-blocking
41 $\quad$ $apl(p)$ **request** $Abort(q_c)$
42 **upon response** $\overline{apl(p'), Commit(q)^{\sigma}}$ s.t. $\{\overline{p'}\} = q_c \wedge q = q_c$ and $\overline{\sigma \text{ is a valid signature of } p'}$
43 $\quad$ $Q \leftarrow Q \cup \{q_n\}$
44 $\quad$ $apl(p'')$ **request** $Success(q_c)^{\{\overline{\sigma}\}}$ for each $p'' \in q_c$
45 $\quad$ **response** *AddComplete*
46 **upon response** $apl(p)$, $Success(q_c)^{\{\overline{\sigma}\}}$ s.t. $\neg succeeded(p, q_c)$ and $\{\overline{\sigma}\}$ are valid signatures of all processes in q_c
47 $\quad$ $succeeded(p, q_c) \leftarrow \mathsf{true}$
48 $\quad$ $apl(p'')$ **request** $Success(q_c)^{\{\overline{\sigma}\}}$ for each $p'' \in q_c$
49 $\quad$ $Q \leftarrow Q \cup \{q_c\}$
50 $\quad$ $tentative \leftarrow tentative \setminus \langle p, q_c \rangle$
51 **upon response** $apl(p'), Abort(q)$ s.t. $q = q_c \wedge p' \in q_c$
52 $\quad$ $apl(p'')$ **request** $Fail(\mathbf{self}, q_c)^{sig}$ for each $p'' \in q_c$
53 $\quad$ **response** *AddFail*
54 **upon response** $apl(p^*)$, $Fail(p, q_c)^{\sigma}$ s.t. $\neg succeeded(p, q_c)$ and σ is a valid signature of p
55 $\quad$ **if** $p^* = p$ **then**
56 $\quad\quad$ $apl(p'')$ **request** $Fail(p, q_c)^{\sigma}$ for each $p'' \in q_c$
57 $\quad$ $failed(p, q_c) \leftarrow failed(p, q_c) \cup \{p^*\}$
58 $\quad$ **if** $q_c \subseteq failed(p, q_c)$ **then**
59 $\quad\quad$ $tentative \leftarrow tentative \setminus \langle p, q_c \rangle$

when q_n is a quorum of p, q_c is either a quorum or a *tentative* quorum of each process in q_c. We now capture this weak notion as tentative quorum inclusion. As we will see, this weaker notion is sufficient to preserve outlived quorum intersection.

Definition 100 (Tentative Quorum inclusion) *A quorum system has* tentative *quorum inclusion for P iff for all well-behaved processes p and quorums q of p, if a process p' in q is inside P, then there is a quorum q' such that well-behaved processes of q' are a subset of q, and q' is in either the quorums set Q or the tentative set of p'.*

We now show that the protocol preserves tentative quorum inclusion for the outlived set $\mathcal{O}$, and eventually restores quorum inclusion for $\mathcal{O}$.

Lemma 101 *The* Add *protocol preserves tentative quorum inclusion. Further, starting from a quorum system that has quorum inclusion for processes $\mathcal{O}$, it eventually results in a quorum system with quorum inclusion for $\mathcal{O}$.*

Availability. Since the protocol does not remove any quorums, it is straightforward that it preserves availability.

Lemma 102 *For every set of processes $\mathcal{O}$, the* Add *protocol preserves quorum availability inside $\mathcal{O}$.*

Quorum Intersection. Now we use the two above properties to show the preservation of quorum intersection.

Lemma 103 *If a quorum system has tentative quorum inclusion for processes $\mathcal{O}$, and availability inside $\mathcal{O}$, then the* Add *protocol preserves quorum intersection at $\mathcal{O}$.*

Outlive. Similar to the Leave and Remove protocols, the three above lemmas show that the Add protocol eventually restores an outlived quorum system, while it always maintains outlived quorum intersection.

Lemma 104 (Preservation of Outlived set) *Starting from a quorum system that is outlived for processes $\mathcal{O}$, the* Add *protocol preserves quorum intersection at $\mathcal{O}$, and eventually results in a quorum system that is outlived for $\mathcal{O}$.*

Immediate from Theorem 103, Theorem 101, and Theorem 102.

Theorem 105 *Starting from a quorum system that is outlived for processes $\mathcal{O}$, the* Add *protocol preserves quorum intersection at $\mathcal{O}$, and eventually provides quorum inclusion for $\mathcal{O}$, and availability inside $\mathcal{O}$.*

Remove and Add. Finally, we note that the Leave and Remove protocols (that we saw at section 4.6) and the Add protocol can be adapted to execute concurrently. The checks for a blocking set are performed in the Leave and Remove protocols (Alg. 9), at line 14 and line 2, and are performed in the Add protocol (Alg. 14) at line 34. The former check considers the *tomb* set and the latter check considers the *pending* set. When the protocols are executed concurrently, both of these sets should be considered. In particular, when the *pending* set is $\{\overline{q'_c}\}$, the check for the former should be that "$\exists q_1, q_2 \in \cup\{\overline{q'_c}\} \cup Q'.\ (q_1 \cap q_2) \setminus (\{p'\} \cup tomb)$ is not p'-blocking", and the check for the latter should be that "$\forall q \in \cup\{\overline{q'_c}\} \cup Q.\ q_c \cap q \setminus tomb$ is **self**-blocking".

4.12 PC Leave and Remove

In this section, we present the Leave and Remove protocols that preserve both consistency and policies. We first consider the protocols before the correctness theorems.

Algorithm 16: PC Leave and Remove a quorum

---

1 **Implements:** Leave and Remove
2 **request** : *Leave* | *Remove*(q)
3 **response** : *LeaveComplete* | *RemoveComplete*
4 **Variables:**
5 Q ▷ The individual minimal quorums of **self**
6
7 F : Set[$\mathcal{P}$]
8 **Uses:**
9 *apl* : $(\cup Q) \cup F \mapsto$ AuthPerfectPointToPointLink
10 **upon request** *Leave*
11 *apl*(p) **request** *Left*(**self**) for each $p \in F$
12 **response** *LeaveComplete*
13 **upon response** *apl*(p), *Left*(p)
14 $Q \leftarrow Q \setminus \{q \in Q \mid p \in q\}$
15 **upon request** *Remove*(q)
16 $Q \leftarrow Q \setminus \{q\}$
17 **response** *RemoveComplete*

The Leave and Remove protocols that preserve the policies are shown in Alg. 16.

Variables and sub-protocols. Each process keeps its own set of individual minimal quorums Q. It also stores its follower processes (*i.e.*, processes that have this process in their quorums) as the set F.

The protocol uses authenticated point-to-point links *apl* (to each quorum member and follower).

Protocol. A process that requests a *Leave* informs its follower set by sending a *Left* message (at line 11). Every well-behaved process that receives a *Left* message (at

line 13) removes any quorum that contains the sender (at line 14) so that quorum intersection is not lost in case the intersection is the leaving process. A process that requests a *Remove*(q) simply removes q locally from its quorums Q.

Correctness. The protocols are both consistency- and policy-preserving.

Lemma 106 *The PC* Leave *and* Remove *protocols are consistency-preserving.*

This is immediate from the fact that the protocols only remove quorums, and further for the leave protocol, the remaining quorums do not include the leaving process. Therefore, quorum intersection persists.

Lemma 107 *The PC* Leave *and* Remove *protocols are policy-preserving.*

This is straightforward as the protocols remove but do not shrink quorums.

4.13 Sink Discovery

Following the graph characterization that we saw in section 4.4, we now present a decentralized protocol that can find whether each process is in the sink component of the quorum graph. We first describe the protocol and then its properties.

Protocol. Consider a quorum system with quorum intersection, availability and sharing. The sink discovery protocol in Alg. 17 finds whether each well-behaved process is in the sink. It also finds the set of its followers. A process p is a follower of process p' iff p has a quorum that includes p'. The protocol has two phases. In the first, it finds the well-behaved minimal quorums, *i.e.*, every minimal quorum that is a subset of well-

behaved processes. Since well-behaved minimal quorums are inside the sink, the second phase extends the discovery to other processes in the same strongly connected component.

Variables and sub-protocols. In the quorum system $\mathcal{Q}$, each process **self** stores its own set of individual minimal quorums $Q = \mathcal{Q}(\mathbf{self})$, a map *qmap* from other processes to their quorums which is populated as processes communicate, the *in-sink* boolean that stores whether the process is in the sink, and the set of follower processes F. The protocol uses authenticated point-to-point links *apl*. They provide the following safety and liveness properties. If the sender and receiver are both well-behaved, then the message will be eventually delivered. Every message that a well-behaved process delivers from a well-behaved process is sent by the later, *i.e.*, the identity of a well-behaved process cannot be forged.

Protocol. When processes receive a *Discover* request (at line 9), they exchange their quorums with each other. In this first phase, each process sends an *Exchange* message with its quorums Q to all processes in its quorums. When a process receives an *Exchange* message (at line 11), it adds the sender to the follower set F, and stores the received quorums in its *qmap*. As *qmap* is populated, when a process finds that one of its quorums q is a quorum of every other process in q as well (at line 14), by Theorem 79, it finds that its quorum q is a minimal quorum, and by Theorem 86, it finds itself in the sink. Thus, it sets its *in-sink* variable to `true` in the first phase (at line 15).

The process then sends an *Extend* message with the quorum q to all processes of its own quorums Q (at line 16). The *Extend* messages are processed in the second phase. The processes of every well-behaved minimal quorum are found in this phase. In Figure 4.3, since the quorum $\{1, 2\}$ is a quorum for both of 1 and 2, they find themselves in the sink.

However, process 3 might receive misleading quorums from process 5, and hence, may not find itself in the sink in this phase.

The processes P_1 of every well-behaved minimal quorum find themselves to be in the sink in the first phase. Let P_2 be the well-behaved processes of the remaining minimal quorums. A pair of minimal quorums have at least a well-behaved process in their intersection. In Figure 4.3, the two minimal quorums $P_1 = \{1, 2\}$ and $P_2 = \{1, 3, 5\}$ intersect at 1. Therefore, by Theorem 80, every process in P_2 is a neighbor of a process in P_1. Thus, in the second phase, the processes P_1 can send *Extend* messages to processes in P_2, and inform them that they are in the sink. In Figure 4.3, process 1 can inform process 3. The protocol lets a process accept an *Extend* message containing a quorum q only when the same message comes from the intersection of q and one of its own quorums q' (at line 17). Let us see why a process in P_2 cannot accept an *Extend* message from a single process. A minimal quorum q that is found in phase 1 can have a Byzantine process p_1. Process p_1 can send an *Extend*(q) message (even with signatures from members of q) to a process p_2 even if p_2 is not a neighbor of p_1, and make p_2 believe that it is in the sink. In Figure 4.3, the Byzantine process 5 can collect the quorum $\{1, 3, 5\}$ from 1 and 3, and then send an *Extend* message to 4 to make 4 believe that it is inside the sink. Therefore, a process p_2 in P_2 accepts an *Extend*(q) message only when it is received from the intersection of q and one of its own quorums. Since there is a well-behaved process in the intersection of the two quorums, process p_2 can then trust the *Extend* message. When the check passes, p_2 finds itself to be in the sink, and sets the *in-sink* variable to `true` in the second phase (at line 18). In Figure 4.3, when process 3 receives an *Extend* message with quorum $\{1, 2\}$ from 1, since

$\{1\}$ is the intersection of the quorum $\{1, 3, 5\}$ of 3, and the received quorum $\{1, 2\}$, process 3 accepts the message.

Algorithm 17: Sink Discovery Protocol

---

1 **Variables:**
2 $\quad Q$ ▷ The individual minimal quorums of **self**
3
4 $\quad qmap : \mathcal{P} \mapsto \mathsf{Set}[2^{\mathcal{P}}]$
5 $\quad \textit{in-sink} : \mathsf{Boolean} \leftarrow \mathsf{false}$
6 $\quad F : 2^{\mathcal{P}}$
7 **Uses:**
8 $\quad apl : \mathcal{P} \mapsto \mathsf{AuthPerfectPointToPointLink}$
9 **upon request** *Discover*
10 $\quad apl(p)$ **request** $\mathit{Exchange}(Q)$ for each $p \in \cup Q$
11 **upon response** $apl(p)$, $\mathit{Exchange}(Q')$
12 $\quad F \leftarrow F \cup \{p\}$
13 $\quad qmap(p) \leftarrow Q'$
14 **upon** $\exists q \in Q.\ \forall p \in q.\ q \in qmap(p)$
15 $\quad \textit{in-sink} \leftarrow \mathsf{true}$
16 $\quad apl(p)$ **request** $\mathit{Extend}(q)$ for each $p \in \cup Q$
17 **upon response** $\overline{apl(p)}, \mathit{Extend}(q)\ \text{s.t.} \exists q' \in Q.\ \{\overline{p}\} = q \cap q'$
18 $\quad \textit{in-sink} \leftarrow \mathsf{true}$

Let *ProtoSink* denote the set of well-behaved processes where the protocol sets the *in-sink* variable to true. The discovery protocol is complete: all the well-behaved processes of minimal quorums will eventually know that they are in the sink (*i.e.*, set their *in-sink* to true).

Lemma 108 (Completeness) *For all* $q \in MQ(\mathcal{Q})$*, eventually* $q \cap \mathcal{W} \subseteq \mathit{ProtoSink}$.

This result brings an optimization opportunity: the leave and remove protocols can coordinate only when a process inside *ProtoSink* is updated. Although, completeness is sufficient for safety of the optimizations, in the appendix section 4.16, we prove both the

completeness and accuracy of the two phases in turn. The accuracy property states that *ProtoSink* is a subset of the sink component.

4.14 AC Leave and Remove Proofs

4.14.1 Remove, Inclusion-preservation

Theorem 93. *The AC* Leave *and* Remove *protocols preserve active quorum inclusion. Further, starting from a quorum system that has quorum inclusion for processes $\mathcal{O}$, the protocols eventually result in a quorum system with quorum inclusion for $\mathcal{O} \setminus \mathcal{L}$.*

Proof. Consider a quorum system $\mathcal{Q}$ with the set of well-behaved processes $\mathcal{W}$. We assume that $\mathcal{Q}$ has quorum inclusion for a set of processes $\mathcal{O}$. Consider a well-behaved process $p_1 \in \mathcal{W}$ and its quorum $q_1 \in \mathcal{Q}(p_1)$, and a process $p_2 \in q_1 \cap \mathcal{O}$ with a quorum $q_2 \in \mathcal{Q}(p_2)$. For active quorum inclusion, we assume $q_2 \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1 \cap \mathcal{W} \setminus \mathcal{L}$, and show that while a process p is leaving or removing a quorum, this property is preserved. For quorum inclusion, we assume $q_2 \cap \mathcal{W} \subseteq q_1 \cap \mathcal{W}$, and show that this properly will be eventually reconstructed.

Consider a process p that receives the response *LeaveComplete* or *RemoveComplete*. Let $\mathcal{L}' = \mathcal{L} \cup \{p\}$ and $\mathcal{O}' = \mathcal{O} \setminus \mathcal{L}'$. We consider four cases. (1) The requesting process is p_1. We consider two cases. (1.a) If p_1 leaves, then it has no quorums. (1.b) If p_1 removes q_1, no obligation for q_1 remains. (2) If the requesting process is p_2, then $p_2 \notin \mathcal{O}'$ and the property trivially holds. (3) If the requesting process is a process p in q_2 such that $p \neq p_2$, then the two processes p_1 and p_2 will receive *Left* messages (at line 10) and will eventually remove p from q_1 and q_2 and result in $q_1' = q_1 \setminus \{p\}$ and $q_2' = q_2 \setminus \{p\}$ respectively. To show active

quorum inclusion, consider that before the two updates, we have $q_2 \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1 \cap \mathcal{W} \setminus \mathcal{L}$. Depending on the order of the two updates, we should show either $q_2' \cap \mathcal{W} \setminus \mathcal{L}' \subseteq q_1 \cap \mathcal{W} \setminus \mathcal{L}'$ or $q_2 \cap \mathcal{W} \setminus \mathcal{L}' \subseteq q_1' \cap \mathcal{W} \setminus \mathcal{L}'$ for the intermediate states, and both trivially hold. After the two updates, we trivially have $q_2' \cap \mathcal{W}' \setminus \mathcal{L}' \subseteq q_1' \cap \mathcal{W} \setminus \mathcal{L}'$. For eventual quorum inclusion, consider the fact that a message from a well-behaved sender is eventually delivered to a well-behaved receiver. Therefore, the quorums q_1 and q_2 will be eventually updated to the eventual states q_1' and q_2' above. Therefore, if $q_2 \cap \mathcal{W} \subseteq q_1 \cap \mathcal{W}$, then we trivially have $q_2' \cap \mathcal{W} \subseteq q_1' \cap \mathcal{W}$. (4) The requesting process $p \neq p_1$ and is in $q_1 \setminus q_2$. The reasoning is similar to the previous case. ■

4.14.2 Remove, Availability-preservation

Theorem 97. *The AC* Leave *and* Remove *protocols preserve active availability. Further, starting from a quorum system that has availability inside processes $\mathcal{O}$, the protocols eventually result in a quorum system with availability inside $\mathcal{O} \setminus \mathcal{L}$.*

Proof. Consider an initial quorum system $\mathcal{Q}$. First, we show that if $\mathcal{Q}$ has active availability inside $\mathcal{O}$, the protocols preserve it. Changes to the quorums of processes in $\mathcal{L}$ does not affect active availability inside $\mathcal{O}$. Other processes can only remove processes in $\mathcal{L}$ from their quorums (at line 11). Therefore, the inclusion of their quorum inside P modulo $\mathcal{L}$ persists. Second, we show that as processes $\mathcal{L}$ leave or remove quorums, the resulting quorum system $\mathcal{Q}'$ will eventually have availability inside $\mathcal{O} \setminus \mathcal{L}$. Consider a process p that is in $\mathcal{O}$ and not $\mathcal{L}$. We show that there will be a quorum $q' \in \mathcal{Q}'(p)$ such that $q' \subseteq \mathcal{O} \setminus \mathcal{L}$. We have that $\mathcal{Q}$ has availability inside $\mathcal{O}$. Thus, $\mathcal{O}$ are well-behaved, and there is a quorum $q \in \mathcal{Q}(p)$ such that $q \subseteq \mathcal{O}$. Let L be the set of well-behaved processes in $\mathcal{L}$. We show

that every p' in L that is in q will be eventually removed from q. Since both p and p' are well-behaved, the *Left* message that p' sends to p (at line 20 or line 9) is eventually delivered to p, and p will remove p' from q (at line 11). Therefore, eventually $q' = q \setminus L$. Thus, since $q \subseteq \mathcal{W}$, $q' = q \setminus \mathcal{L}$. Thus, since $q \subseteq \mathcal{O}$, we have $q' \subseteq \mathcal{O} \setminus \mathcal{L}$

■

4.14.3 Remove, Intersection-preservation

Theorem 98. *If a quorum system has quorum intersection at processes $\mathcal{O}$, active availability inside $\mathcal{O}$, and active quorum inclusion for $\mathcal{O}$, then the AC* Leave *and* Remove *protocols preserve quorum intersection at $\mathcal{O} \setminus \mathcal{L}$.*

Proof.

Assume that a quorum system $\mathcal{Q}$ has quorum intersection at processes $\mathcal{O}$, availability inside $\mathcal{O}$ and active quorum inclusion for $\mathcal{O}$. We have two cases for the requesting process **self**: it is either in $\mathcal{O}$ or not. In the latter case, it cannot affect the assumed intersection at $\mathcal{O}$.

Now let us consider the case where the leaving process **self** is in $\mathcal{O}$. Consider two well-behaved processes p_1 and p_2 with quorums $q_1 \in \mathcal{Q}(p_1)$ and $q_2 \in \mathcal{Q}(p_2)$. Let I be the intersection of q_1 and q_2 in $\mathcal{O}$, *i.e.*, $I = q_1 \cap q_2 \cap \mathcal{O}$. Assume that **self** is in the intersection of q_1 and q_2, *i.e.*, **self** $\in I$. We assume that **self** receives a *LeaveComplete* or *RemoveComplete* response. We show that the intersection of the two quorums has another process in $\mathcal{O}$. Let L be the subset of processes in I that have received a *LeaveComplete* or or *RemoveComplete* response before **self** receives hers. After the processes $\mathcal{L}$ and **self** receive a response, the quorums will incrementally shrink (at line 11) where the final smallest

quorums are $q'_1 = q_1 \setminus (\mathcal{L} \cup \{\mathbf{self}\})$ and $q'_2 = q_2 \setminus (\mathcal{L} \cup \{\mathbf{self}\})$ respectively. Let $\mathcal{L}' = \mathcal{L} \cup \{\mathbf{self}\}$ and $\mathcal{O}' = \mathcal{O} \setminus \mathcal{L}'$. We show that even the smallest quorums have intersection in $\mathcal{O}'$, *i.e.*, $q'_1 \cap q'_2 \cap \mathcal{O}' \neq \emptyset$.

A *LeaveComplete* or *RemoveComplete* response is issued (at line 8) when processing a *Check* request. The total-order broadcast *tob* totally orders the *Check* deliveries. Let p^* be the process in $(L \cup \{\mathbf{self}\})$ that is ordered last in the total order. By Theorem 93, active quorum inclusion is preserved. Therefore, since p^* is in q_1 and $\mathcal{O}$, there is a quorum $q^*_1 \in \mathcal{Q}(p^*)$ such that $q^*_1 \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1 \cap \mathcal{W} \setminus \mathcal{L}$. Similarly, we have that there is a quorum $q^*_2 \in \mathcal{Q}(p^*)$ such that $q^*_2 \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_2 \cap \mathcal{W} \setminus \mathcal{L}$. Since p^* has received a complete response, $q^*_1 \cap q^*_2 \setminus (\{p^*\} \cup \mathit{tomb})$ is p^*-blocking (by the condition at line 2). The total-order broadcast *tob* ordered the *Check* deliveries for every process in the set $L \cup \{\mathbf{self}\} \setminus \{p^*\}$ before the *Check* delivery for p^*. Therefore, since every process that gets a complete response is added to the *tomb* set (at line 6), the *tomb* set of p^* includes these processes, *i.e.*, $L \cup \{\mathbf{self}\} \setminus \{p^*\} \subseteq \mathit{tomb}$. Therefore, by substitution of *tomb*, we have $q^*_1 \cap q^*_2 \setminus (L \cup \{\mathbf{self}\})$ is a blocking set for p^*. By the definition of L above, we have that the processes $\mathcal{L} \setminus L$ are not in the intersection of q_1, q_2 and $\mathcal{O}$. Therefore, $q^*_1 \cap q^*_2 \setminus (\mathcal{L} \cup \{\mathbf{self}\})$ is a blocking set for p^*. Therefore, $q^*_1 \cap q^*_2 \setminus \{\mathbf{self}\}$ is an active blocking set for p^*. By Theorem 96, there is a process $p \in \mathcal{O} \setminus \mathcal{L}$ such that $p \in q^*_1 \cap q^*_2 \setminus (\{\mathbf{self}\})$. We have $(q^*_1 \cap q^*_2 \setminus (\{\mathbf{self}\})) \cap (\mathcal{O} \setminus \mathcal{L}) \neq \emptyset$. Distribution of $\setminus$ give us $(q^*_1 \setminus \mathcal{L} \cup \{\mathbf{self}\}) \cap (q^*_2 \setminus \mathcal{L} \cup \{\mathbf{self}\}) \cap (\mathcal{O} \setminus \mathcal{L} \cup \{\mathbf{self}\}) \neq \emptyset$. By active quorum inclusion, we have $q^*_1 \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1 \cap \mathcal{W} \setminus \mathcal{L}$ and $q^*_2 \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_2 \cap \mathcal{W} \setminus \mathcal{L}$. Thus, we have $(q_1 \setminus \mathcal{L} \cup \{\mathbf{self}\}) \cap (q_2 \setminus \mathcal{L} \cup \{\mathbf{self}\}) \cap (\mathcal{O} \setminus \mathcal{L} \cup \{\mathbf{self}\}) \neq \emptyset$. Thus, $q'_1 \cap q'_2 \cap \mathcal{O}' \neq \emptyset$.

■

4.15 Add Proofs

4.15.1 Add, Inclusion-preservation

Theorem 101. *The* Add *protocol preserves tentative quorum inclusion. Further, starting from a quorum system that has quorum inclusion for processes $\mathcal{O}$, it eventually results in a quorum system with quorum inclusion for $\mathcal{O}$.*

Proof. Consider a quorum system $\mathcal{Q}$ with the set of well-behaved processes $\mathcal{W}$. Consider a set of well-behaved processes $\mathcal{O}$, a well-behaved process $p_1 \in \mathcal{W}$ and its quorum $q_1 \in \mathcal{Q}(p_1)$, and a process $p_2 \in q_1 \cap \mathcal{O}$ with a quorum $q_2 \in \mathcal{Q}(p_2)$. For tentative quorum inclusion, we assume $q_2 \cap \mathcal{W} \subseteq q_1 \cap \mathcal{W}$, and that q_2 is in either $\mathcal{Q}(p_2)$ or the *tentative* set of p_2, and show that while a process p is adding a quorum, this property is preserved.

We consider a case for each such line where a quorum is added.

Case (1): The quorum q_n is added at line 26. By line 22-line 23, and line 25, the process p has received the *AckInclusion* message from every process $p' \in q_n$. Since before sending *AckInclusion*, a well-behaved p' makes sure it has a quorum inside q_n (at line 16) and $\mathcal{O} \subseteq \mathcal{W}$, we have $\forall p' \in q_n \cap \mathcal{O}.\ \exists q' \in \mathcal{Q}(p').\ q' \subseteq q_n$. This is sufficient for quorum inclusion that is stronger than tentative quorum inclusion.

Case (2): The quorum q_n is added at line 43: The sets $q_n \setminus q_c$ and q_c are the processes in q_n which, respectively, do and do not satisfy quorum inclusion with their existing quorums (line 16-19 and line 22-23). Further, $q_c \subseteq q_n$. Therefore, we only need to show tentative quorum inclusion for processes p' in q_c. Before adding q_n, the process p receives the *Commit* message from every process $p' \in q_c$ (at line 42). Before sending the *Commit* message, every well-behaved process $p' \in q_c$ receives *CheckAck* messages from all

the processes of one of its quorums including itself (at line 38). p_o only send *CheckAck* message after receives *Check* message from p' at line 31, which is after q_c has been added to *tentative* of p' Therefore, (a) before q_n is added as a quorum of p, every well-behaved process p' in q_c adds q_c to its *tentative* set. We will show below that (b) every well-behaved process p' in q_c, removes q_c from the *tentative* set only after it is already added to its set of quorums. Since $\mathcal{O} \subseteq \mathcal{W}$, (a) and (b) above show tentative quorum inclusion for $\mathcal{O}$.

We now show the assertion (b) above. p' removes q_c from its *tentative* set only after it receives *Success* or *Fail* messages. We consider a case for each Case (2.1): q_c is removed from *tentative* at line 50. q_c is removed from *tentative* only after q_c is added to the set of quorums Q. Case (2.2): q_c is removed at line 59, which is after process p' receiving *Fail* messages from every member of q_c (at line 58) and verifying the signature from p (at line 54). However, the process p has sent a *Success* message (at line 44) after q_n is added. If a well-behaved process sends a *Success* message, then it returns the response *AddComplete* and the add process finishes. Therefore, since p is a well-behaved process, it does not send a *Fail* message and this case does not happen.

Case (3): The quorum q_c is added at line 49. This add happens only after the current process p' receives a *Success* message (at line 46) . Upon delivery of a *Success* message, p' validates a signature from every process in q_c for *Commit*. Therefore, by an argument similar to Case (2), we have that (a) before a process adds q_c to its quorums, every well-behaved process in q_c has added q_c to its *tentative* set. We will show below that (b) every well-behaved process in q_c removes q_c from the *tentative* set only after it is already added to its set of quorums. Since $\mathcal{O} \subseteq \mathcal{W}$, (a) and (b) above show tentative quorum

inclusion for $\mathcal{O}$. We now show the assertion (b) above. A process p'' removes q_c from its *tentative* set only after delivery of *Success* or *Fail* messages and we consider a case for each. Case (3.1): q_c is removed at line 50. The argument is similar to Case (2.1). Case (3.2): q_c is removed at line 59, which is after receiving the *Fail* messages from all the members of q_c including p'. However, before q_c is added to the quorums of p', p' set *succeeded* to `true` at line 47. The condition $\neg \mathit{succeeded}(p, q_c)$ at line 54 prevents it from sending *Fail* after receiving *Success*. Therefore, since p' has received a *Success* message, then it does not send a *Fail* message. Therefore, p'' can not remove q_c from its *tentative* at line 59.

For eventual quorum inclusion, consider the requesting process p that adds q_n to its quorums. The argument is similar to the Case (1) and Case (2) above for tentative quorum inclusion. In the second case, before adding q_n (at line 43), the process p sends the *Success* message to processes in q_c (at line 44). Therefore, all the well-behaved processes in q_c eventually add q_c to their set of quorums (at line 49). Therefore, q_c and q_n will eventually have quorum inclusion. ■

4.15.2 Add, Availability-preservation

Theorem 102. *For every set of processes $\mathcal{O}$, the* Add *protocol preserves quorum availability inside $\mathcal{O}$.*

Proof. The Add protocol does not explicitly remove any quorums. An implicit removal can happen when a process has a quorum that is a subset of another. We show that if the addition of quorum q_c leads to an implicit removal of a quorum, availability is preserved.

If q_c is a subset of an existing quorum, then even if either of the quorums is removed, availability is preserved. Further, q_c is not a superset of any quorum q of a process p in q_c. Otherwise, q is a subset of q_n and the process p would not be included in q_c. ■

4.15.3 Add, Intersection-preservation

Theorem 103. *If a quorum system has tentative quorum inclusion for processes $\mathcal{O}$, and availability inside $\mathcal{O}$, then the* Add *protocol preserves quorum intersection at $\mathcal{O}$.*

Proof. Consider well behaved processes $\mathcal{O}$, and a quorum system $\mathcal{Q}$ with tentative quorum inclusion for $\mathcal{O}$, availability inside $\mathcal{O}$, and quorum intersection at $\mathcal{O}$. Consider a well-behaved process p that requests $Add(q_n)$. The new quorums that are added to the quorum system are q_c and its superset q_n. Consider a well-behaved process p_w with a quorum q_w that is either an existing quorum or a tentative quorum that passed the all the checks. We should show that both q_c and q_n intersect q_w at $\mathcal{O}$.

We assumed that there is a well-behaved process p' in q_n. We consider two cases: Case (1) The well-behaved process p' is in q_c. A process adds q_c (at line 49) only after receiving a *Success* message (at line 46). The process p sends a *Success* message (at line 44) only after receiving the *Commit* message form every process in q_c (at line 42). The well-behaved process p' sends a *Commit* message (at line 39) only after receiving a *CheckAck* message from one of its quorums q' (at line 38). By quorum intersection at $\mathcal{O}$, the intersection of q' and q_w has a process p_o in $\mathcal{O}$. By tentative quorum inclusion for $\mathcal{O}$, there is a quorum q_o of p_o such that $q_o \cap \mathcal{W} \subseteq q_w \cap \mathcal{W}$ and either q_o is a quorum of p_o or a member of its *tentative* set. Since p_o has sent an *CheckAck* message (at line 35), it has passed the check that $q_o \cap q_c$ is p_o-blocking (at line 33-line 34). Since $\mathcal{O}$ is available and p_o is in $\mathcal{O}$, then by Theorem 75,

we have that $q_o \cap q_c \cap \mathcal{O} \neq \emptyset$. Since $q_o \cap \mathcal{W} \subseteq q_w \cap \mathcal{W}$, and $\mathcal{O} \subseteq \mathcal{W}$, then $q_w \cap q_c \cap \mathcal{O} \neq \emptyset$. Since $q_c \subseteq q_n$, we have $q_w \cap q_n \cap \mathcal{O} \neq \emptyset$.

Case (2) The well-behaved process p' is in $q_n \setminus q_c$. A process in q_n is not in q_c only if it already satisfies quorum inclusion. (line 16-19 and line 22-23). Therefore, p' has a quorum q such that $q \subseteq q_n$. Thus, the quorum intersection for q implies quorum intersection for q_n. If q_c has a well-behaved process, the proof follows the previous case. Otherwise, it has no well-behaved process, and quorum intersection is only required for well-behaved processes. ■

4.16 Sink Discovery Proofs

4.16.1 Sink Discovery, Completeness

Let the set *ProtoSink* partition into $ProtoSink_1$ and $ProtoSink_2$ that denote the set of well-behaved processes at which the protocol sets the *in-sink* variable to `true` in phase 1 (at line 15), and in phase 2 (at line 18) respectively.

Well-behaved minimal quorums will eventually be in $ProtoSink_1$.

Lemma 109 (Completeness for Phase 1) *./C4/ProtoSink1Completeness*

Proof. Consider a well-behaved minimal quorum q. By Theorem 79, every process in q is in a quorum of every other process in q. Since all processes in q are well-behaved, they send out *Exchange* messages to all the other processes in q (at line 10). Since these processes are well-behaved, they will eventually receive each other's messages, and record each other quorums (at line 11). Therefore, each of them will eventually satisfy the condition, and set *in-sink* to `true` (at line 15-line 14). ■

Well-behaved processes in the minimal quorums will eventually be in either $ProtoSink_1$ or $ProtoSink_2$.

Lemma 110 (Completeness for Phase 2) *Forall* $q \in MQ(\mathcal{Q})$*, eventually* $q \cap \mathcal{W} \subseteq ProtoSink_1$ *or* $q \cap \mathcal{W} \subseteq ProtoSink_2$*.*

Proof. Consider a minimal quorum q which is not found in phase 1, *i.e.*, it is not added to $ProtoSink_1$. Since the quorum system is available (for a set of processes), there exists a well-behaved quorum q'. By the consistency property, q and q' have an intersection

$\{\overline{p}\}$. Since q' is well-behaved, $\{\overline{p}\}$ are well-behaved. Since each process p in $\{\overline{p}\}$ is in q', by Theorem 109, p sets its *in-sink* variable to `true` (at line 15), and then sends out *Extend* messages to its neighbors: all processes of its individual minimal quorums (at line 16). Since each process p is in q, by Theorem 80, every well-behaved process of q is a neighbor of p. Therefore, all well-behaved processes in q will receive the *Extend* messages from the intersection $\{\overline{p}\}$, and set their *in-sink* variable to `true`. ■

4.16.2 Sink Discovery, Accuracy

We saw above that completeness is sufficient for safety of the optimizations. We now consider the accuracy property: every well-behaved process that sets its *in-sink* variable to `true` is in the sink component. Let us consider an attack scenario for accuracy. A group of Byzantine processes fake to be a quorum and send an *Extend* message to make a process that it is outside the sink believe that it is in the sink. This can violate accuracy. Therefore, a check *validq* is needed (at line 17) to ensure that the processes in q are a valid quorum in the system. For example, the heterogeneous quorum systems of both Stellar [307] and Ripple [397] use hierarchies of processes and size thresholds to recognize quorums. We prove the accuracy of the two phases in turn that results in the following lemma. Let $Sink(\mathcal{Q})$ denote processes in the sink component of $\mathcal{Q}$.

Lemma 111 (Accuracy) $ProtoSink \subseteq Sink(\mathcal{Q})$.

Lemma 112 (Accuracy of Phase 1) $ProtoSink_1 \subseteq Sink(\mathcal{Q})$

Proof. Processes in $ProtoSink_1$ set *in-sink* to `true` (at line 15) after they check

(at line 14) that one of their quorums q in an individual minimal quorum of all its members. By Theorem 79, q is a minimal quorum, and by Theorem 86, q is in the sink. ■

Lemma 113 (Accuracy for Phase 2) $\mathit{ProtoSink}_2 \subseteq \mathit{Sink}(\mathcal{Q})$

Proof. Consider a process p^* in $\mathit{ProtoSink}_2$. It sets *in-sink* to `true` (at line 18) only after receiving an *Extend* message containing a quorum q from a set of processes $P' = \{\overline{p'}\}$ such that P' is the intersection of q and a quorum of p^* (at line 17). Further, the *validq* check ensures that q has at least one well-behaved process p_w (at line 17). Since well-behaved processes only send *Extend* messages with a minimal quorum, q is a minimal quorum. Since p^* receives a message from p_w, there is an edge from p_w to p^* in the quorum graph. We show that there is a path from p^* to p_w. By Theorem 81, there are edges from p^* to all members of at least one minimal quorum q'. By consistency, there is at least one well-behaved process p'_w in the intersection of q' and q. There is an edge from p^* to p'_w. By Theorem 82, there is an edge from p'_w to p_w. Thus, there is a path from p^* to p_w though p'_w. Therefore, they are strongly connected. By Theorem 112, p_w is in the sink, Therefore, p^* is in the sink as well. ■

4.17 Discussion

We contrast HQS from ABQS on the following two aspects: 1) ABQS is based on asymmetric failure prone sets F_i from each process p_i. The notions of consistency and availability are dependent on the F_i sets . In contrast, HQS, and its consistency and availability properties are not dependent on failure prone sets. 2) We say that a set of processes is strongly available if there is a well-behaved quorum with inclusion for each process in that set. A maximal strongly available set (from HQS) is a strict superset of a maximal guild (from ABQS): all the maximal guilds in ABQS are strongly available in HQS with quorum intersection; however, the opposite is not true. We show a quorum system that has an empty guild but has non-empty strongly available set, and quorum intersection. Let $\mathcal{P} = \{1, 2, 3, 4\}$, and process 3 is Byzantine. Let the asymmetric failure prone system be $\mathcal{F}(1) = \{\{3, 4\}, \{2\}\}$, $\mathcal{F}(2) = \{\{3, 4\}, \{1\}\}$, $\mathcal{F}(3) = \{\{1, 2\}\}$, $\mathcal{F}(4) = \{\{2, 3\}\}$. The canonical quorum system is $\mathcal{Q}(1) = \{\{1, 2\}, \{1, 3, 4\}\}$, $\mathcal{Q}(2) = \{\{1, 2\}, \{2, 3, 4\}\}$, $\mathcal{Q}(3) = \{\{3, 4\}\}, \mathcal{Q}(4) = \{\{1, 4\}\}$. This HQS has quorum intersection and a non-empty strongly available set: all the quorums of well-behaved processes have at least one correct process in their intersection. The set $\{1, 2\}$ is a strongly available set for $\mathcal{Q}$. However, it is not an ABQS with generalized B^3 condition. For processes 1 and 2, $\mathcal{F}_1 = \{2\}, \mathcal{F}_2 = \{1\}$ and $\mathcal{F}_{12} = \{3, 4\}$, and we have $\mathcal{P} \subseteq \{2\} \cup \{1\} \cup \{3, 4\}$, which violates generalized B^3. Therefore this quorum system is not an ABQS. There is no guild set.

4.18 Related Works

Quorum Systems with Heterogeneous Trust. We described a few instances of heterogeneous quorum systems in section 4.3. The blockchain technology raised the interest in quorum systems that allow non-uniform trust preferences for participants, and support open admission and release of participants. Ripple [397] and Cobalt [319] pioneered decentralized admission. They let each node specify a list, called the unique node list (UNL), of processes that it trusts. However, they assume that 60-90% of every pair of lists overlap. It has been shown that violation of this assumption can compromise the security of the network [32, 434], further highlighting the importance of formal models and proofs [218, 78].

Stellar [334] provides a consensus protocol for federated Byzantine quorum systems (FBQS) [185, 187] where nodes are allowed to specify sets of processes, called slices, that they trust. The Stellar system [307] uses hierarchies and thresholds to specify quorum slices and provides open membership. Since each process calculates its own quorums from slices separately, the resulting quorums do not necessarily intersect, and after independent reconfigurations "the remaining sets may not overlap, which could cause network splits" [148]. Therefore, to prevent forks, a global intersection check is continually executed over the network. Follow-up research analyzed the decentralization extent of Stellar [85, 244], and discussed [180] reconfiguration for the uniform quorums of the top tier nodes. This paper presents reconfiguration protocols that preserve the safety of heterogeneous quorum systems.

Personal Byzantine quorum systems (PBQS) [311] capture the quorum systems that FBQSs derive from slices, require quorum intersection only inside subsets of processes called clusters, and propose a consensus protocol. It defines the notion of quorum sharing. As we saw in section 4.3, this paper presents quorum inclusion that is weaker than quorum sharing; therefore, a cluster is outlived but not vice versa. This paper showed that quorum inclusion is weak enough to be preserved during reconfiguration, and strong enough to support preserving consistency and availability.

Flexible BFT [325] allows different failure thresholds between learners. Heterogeneous Paxos [401, 402] further generalizes the separation between learners and acceptors with different trust assumptions. Further, it specifies quorums as sets rather than number of processes. These two projects introduce consensus protocols. However, they require the knowledge of all processes in the system. In contrast, this paper presents HQSs that requires only local knowledge, captures their properties, and presents reconfiguration protocols for them.

Asymmetric trust [142] lets each process specify the sets of processes that its doesn't trust, and considers broadcast, secret-sharing, and multi-party computation problems. Similarly, in asymmetric Byzantine quorum systems (ABQS) [104, 105, 25] each process defines its subjective dissemination quorum system (DQS): in addition to its sets of quorums, each process specifies sets of processes that it believes may mount Byzantine attacks. This work presents shared memory and broadcast protocols, and further, rules to compose two ABQSs. The followup model [103] lets each process specify a subjective DQS for processes that it knows, transitively relying on the assumptions of other processes.

On the other hand, this paper presents decentralized reconfiguration protocols to add and remove processes and quorums. Further, it lets each process specify only its own set of quorums, and captures the properties of the resulting quorum systems.

Multi-threshold [223] and MT-BFT [341] broadcast protocols elaborate Bracha [86] to have different fault thresholds for different properties and for different synchrony assumptions but have uniform quorums. K-CRB [70] supports non-uniform quorums and delivers up to k different messages.

Quorum subsumption [289] adopted and cited HQS from the arXiv version of this paper, and presented a generalization of quorum sharing called quorum subsumption. This paper focuses on maintaining the properties of HQS when processes perform reconfigurations. We found quorum inclusion as a flavor of quorum sharing that is weak enough to be maintained during reconfiguration, and strong enough to support the consistency and availability properties. In fact, quorum inclusion is weaker than quorum subsumption: for a quorum q, it requires only the processes of q that are in P to have a quorum q', and only the well-behaved part of q' to be a subset of q.

Open Membership. We consider three categories.

Group Membership. As processes leave and join, group membership protocols [128] keep the same global view of members across the system (although the set of all processes may be fixed). Pioneering work, Rambo [317] provides atomic memory, and supports join and leave reconfigurations. It uses Paxos [263] to totally order reconfiguration requests, and tolerates crash faults. Rambo II [195] improves latency by garbage collecting in parallel. Recently, multi-shard atomic commit protocols [89] reduce the number of replicas for each

shard and reconfigure the system upon failures. Since accurate membership is as strong as consensus [121, 128], classical [383] and recent Byzantine group membership protocols such as Cogsworth [351] and later works [191] use consensus to reach an agreement on membership and adjust quorums accordingly. Recent works present more abstractions on top of group membership: DBRB [205] provides reliable broadcast, DBQS [29] preserves consistent read and write quorums, and further adjusts the cardinality of quorums according to the frequency of failures, Dyno [158] provides replication, and SmartMerge [233] provides replication, and uses a commutative merge function on reconfiguration requests to avoid consensus. Existing protocols consider only cardinality-based or symmetric quorum systems where quorums are uniform across processes. On the other hand, this paper presents reconfiguration protocols for heterogeneous quorum systems.

Hybrid Open Membership. Solida [15], Hybrid Consensus [366], Tendermint [94, 33], Casper [100], OmniLedger [247], and RapidChain [449] blockchains combine permissionless and permissioned replication [323] to provide both consistency and open membership [55]. They use permissionless consensus to dynamically choose validators for permissioned consensus.

Unknown participants and network topology. BCUP and BFT-CUP [117, 23, 22] consider consensus in environments with unknown participants. They assume properties about the topology and connectivity of the network, and consider only uniform quorums. Later, [354] presents necessary and sufficient conditions for network connectivity and synchrony for consensus in the presence of crash failures and flaky channels. In contrast,

this paper considers reconfiguration for Heterogeneous Byzantine quorums, and presents optimizations based on the quorum topology.

4.19 Conclusion

This paper presents a model of heterogeneous quorum systems, their properties, and their graph characterization. In order to make them open, it addresses their reconfiguration. It proves trade-offs for the properties that reconfigurations can preserve, and presents reconfiguration protocols with provable guarantees, We hope that this work further motivates the incorporation of open membership and heterogeneous trust into quorum systems, and helps blockchains avoid high energy consumption, and centralization at nodes with high computational power or stake.

Chapter 5

Reconfigurable Clustered Byzantine Replication

5.1 Introduction

Blockchains such as Bitcoin [350] and Ethereum [435] maintain a global replicated ledger on untrusted hosts. However, they suffer from a few drawbacks including high energy consumption, partitions [416, 192, 391], and stake and vote centralization [428]. Byzantine replicated systems such as PBFT [116] and its numerous following variants [338, 445, 410, 35, 411] can maintain consistent replications in the presence of malicious nodes. More interestingly, these techniques require nodes to use a *modest amount of energy*. Therefore, they are an appealing technology to serve as the global financial infrastructure. Thus, several projects such as Hyperledger [35], Solida [15], Tendermint [94], Casper [100], Algorand [194],

OmniLedger [247] and RapidChain [449] deployed Byzantine replication protocols to manage blockchains.

However, Byzantine replication protocols need to be improved on two fronts: *scale* and *openness*. They often require rounds of message-passing between nodes; therefore, they tend not to scale on many or distant nodes. Further, their membership is often closed. In fact, the resulting blockchains are called permissioned since their set of nodes is fixed and initially known.

To scale Byzantine replication across the globe, projects such as Steward [30] and ResilientDB [212] try to use global communication judiciously, and decrease global in favor of local communication. In contrast to monolithic replicated systems, they let adjacent nodes form clusters, and let each cluster locally order transactions, and then only briefly communicate the orders globally to maintain agreement. Since the communication between members of a cluster is local, clusters can maintain high throughput and low latency. Further, coordination is divided between clusters, and they can order transactions in parallel.

However, existing clustered replication protocols are homogeneous and closed. The number of nodes is the same across clusters. Further, nodes cannot join or leave clusters. A global financial system needs to be *heterogeneous*: different regions might have different number of active nodes. More importantly, decentralization promised *openness*: active nodes should be able to churn. This is the property that proof-of-work blockchains such as Bitcoin live on: unknown incentivized servers can join and keep it running. Reconfiguration has been studied for monolithic replication [158, 233, 156, 266, 262, 204] but is an open

problem for clustered replication. Can we have the best of both worlds? *Can we have the energy efficiency, equity and scalability of clustered Byzantine replication, and the openness of proof-of-work? Can we have reconfigurable clustered replication?*

Reconfiguring a clustered replication system *without compromising security* is a challenging task. If the reconfigurations are not propagated uniformly to all clusters, correct processes might accept fake messages or miss genuine ones. Thus, inconsistent views of membership may lead to violation of both safety and liveness. Byzantine replicated systems can often tolerate one-third of processes to be Byzantine. Thus, if a message is received from more than one-third of processes, at least one correct process must have sent it; therefore, the message can be trusted. Consider a cluster C that is not informed of new additions to another cluster C'. The cluster C's record of one-third is less than the actual one-third for C'. Therefore, the Byzantine processes in C' can form a group that is larger than the old one-third, and can make C accept a fake message. On the flip side, C' might miss messages from C. Since C thinks that C' is smaller, in order to communicate a message, C might send a message to an insufficient number of processes in C'. Thus, the Byzantine processes in C' can censor the message for other processes in that cluster, and endanger liveness. Uniform propagation of reconfigurations is particularly challenging when the leader of the cluster simultaneously changes.

In this paper, we present reconfigurable clustered Byzantine replication: we describe a Byzantine replication protocol that allows nodes to be divided into multiple *heterogeneous* clusters, and further *join and leave* clusters. Before a brief communication with other clusters in each round, each cluster orders transactions independently of other clus-

ters, and further accepts join and leave requests. Reconfigurations are processed efficiently in parallel to transactions. Since the reconfigurations received in a round take effect for the next round, they do not need to be ordered in a round. Thus, instead of processing them through consensus in sequence, they are *aggregated* as a set, and processed together. We will present the reconfiguration protocol and formally state and prove its safety and liveness.

Heterogeneity that reconfiguration brings to clusters complicates the processing of transactions. In particular, the protocols that broadcast across clusters, and remotely change leaders, have to keep a consistent record of the size of the local and remote clusters, and send messages accordingly. In addition to reconfiguration, the clustered replication protocol that we present in this paper maintains replication across heterogeneous clusters.

The reconfigurable clustered replication protocol is *parametric* in terms of the local replication protocol. We implement the protocol for HotStuff [445] in C++, and for BFT-SMaRt [67] in Java. We deployed the resulting systems on geo-distributed clusters in multiple regions of Google cloud. The experimental results show that heterogeneous geo-distributed deployments significantly improve throughput, can be reconfigured without affecting transaction processing, and can gracefully tolerate Byzantine failures.

In short, this paper makes the following contributions:

- *Reconfiguration protocol for clustered replication.* The protocol allows processes to join and leave clusters safely and efficiently.

- *Heterogeneous clustered replication.* Clusters can have different sizes, and the inter-cluster broadcast, and remote leader change protocols adapt to different sizes.

- *Statement and proof of safety and liveness* properties for the reconfiguration protocol.
- *Implementation and empirical results.* The protocol is parametric for the local replication protocol, and we instantiated it for both HotStuff and BFT-SMaRt. Experiments show that heterogeneity improves performance, and the resulting replicated system are reconfigurable and fault-tolerant.

We start with an overview.

5.2 Overview

In this section, we describe the system and threat model, and illustrate the protocol with diagrams and representative executions.

System and Threat Model. A replicated system consists of a set $\mathcal{P}$ of processes that are partitioned into clusters $\mathcal{C} = \{C_1, .., C_N\}$. Clients can send requests to any process to execute operations of two different types: transactions and reconfigurations. The state is replicated at each process. (In contrast to sharding, the state is not partitioned.) A process can be correct or Byzantine. A Byzantine process can fail arbitrarily including but not limited to crash failures, sending conflicting messages, dropping messages, and impersonating other Byzantine processes. We assume that at any time in each cluster, at most one-third of processes can be Byzantine, *i.e.*, at most f out of $3f + 1$ processes can be Byzantine. (This paper does not consider problems orthogonal to Byzantine fault tolerance such as access control or sybil resistance [378, 417, 454].) We further assume that each process can be identified by its public key, and that processes are computationally bound,

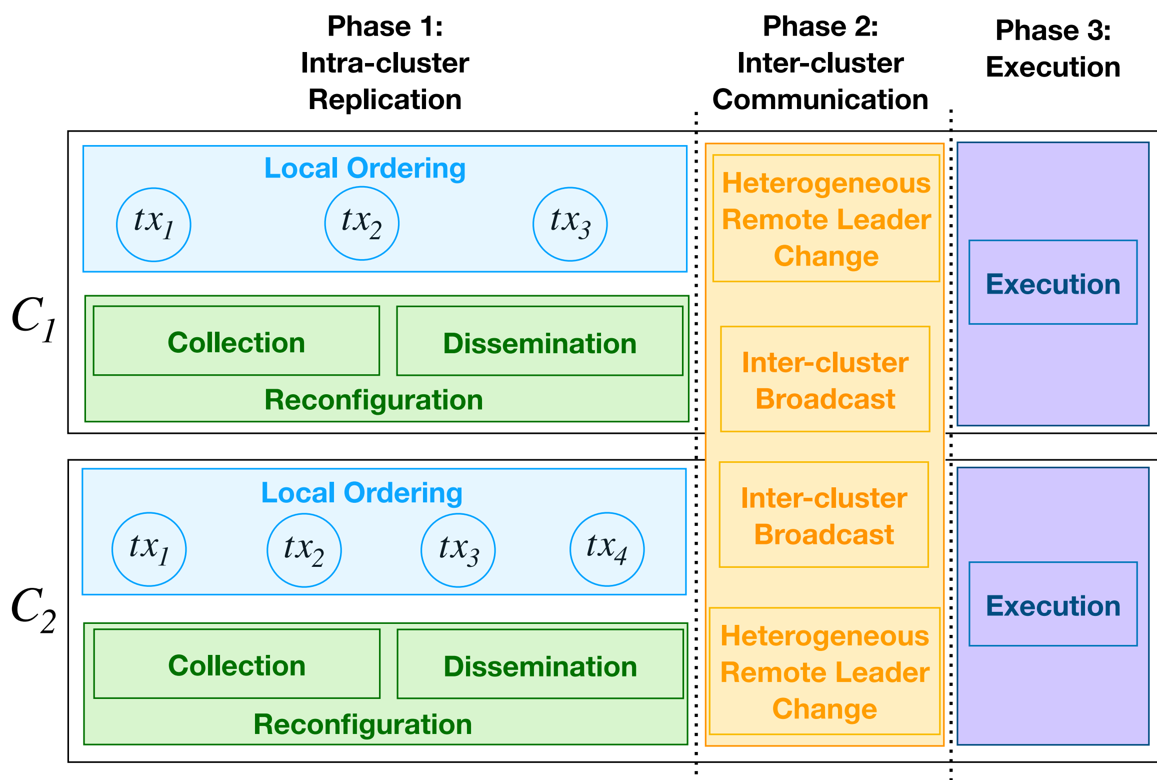


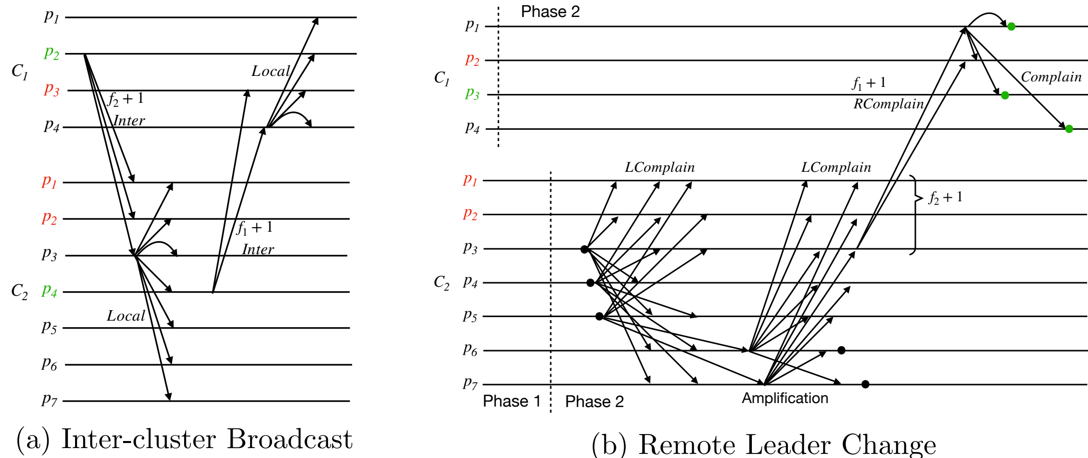
Figure 5.1: Overview of Phases and Sub-protocols

and cannot subvert standard cryptographic primitives. Thus, processes can communicate with authenticated links. We consider a partially synchronous network [160]: after an unknown global stabilization time, messages between any pair of correct processes will be eventually delivered within a bounded delay. Processes communicate with authenticated perfect links *apl*, and authenticated best-effort broadcast *abeb* which simply abstracts *apl* to send a message to all processes. Each message m^σ delivered from an authenticated link comes with a signature σ of the sender. (We elide the signature when it is not needed in a context.)

Each cluster has a leader that coordinates the replication of both transactions and reconfigurations. Each process stores the *leader* of its cluster and its associated timestamp ts. The protocol proceeds in consecutive rounds r. A leader might continue to serve for multiple rounds. On the other hand, several leaders might change within a round until a correct leader properly replicates transactions and reconfigurations.

Phases. Each round has three phases. In each round, a batch of transactions from each cluster is executed. Figure 5.1 shows an overview of the phases and sub-protocols in a round that we elaborate in this and next sections. The processes are split into clusters. The figure shows two clusters C_1 and C_2 that chronologically make progress from left to right through the phases. The first phase is intra-cluster replication where each cluster coordinates replication locally and independently of other clusters. The first phase has two parts that are executed in parallel: local ordering, and reconfiguration. The local ordering protocol orders a batch of transactions uniformly across the processes of the cluster. The protocol is parametric in terms of the local ordering sub-protocol; any monolithic replication

protocol can be used. The second part, the reconfiguration protocol, collects and uniformly disseminates the reconfiguration requests across the cluster, even if the leader is Byzantine or changes simultaneously. After the first phase finishes intra-cluster replication, the second phase performs inter-cluster communication: the leader of each cluster broadcasts to other clusters the transactions and reconfigurations that it has locally replicated. Each cluster waits to receive these messages from every other cluster. If a remote leader is Byzantine, it may refrain from sending these messages. Therefore, to ensure progress, if the processes of a cluster don't receive the message from a remote cluster, they trigger the remote leader change protocol to eventually change the leader of that remote cluster. Finally, in the third phase, each process orders the transactions and reconfigurations that it has received from all clusters by a predefined order for the clusters, executes them in order, and issues responses. This predefined order yields a total-order for operations across replicas. Processes converge to the same state at the end of the round.



(a) Inter-cluster Broadcast

(b) Remote Leader Change

Figure 5.2: Phase 2: Inter-cluster Communication. $f_1 = 1$, $f_2 = 2$.

Inter-cluster Communication. Let us consider the inter-cluster communi-

cation phase (*i.e.*, phase 2). Figure 5.2 shows example executions of this phase for two heterogeneous clusters C_1 and C_2 with 4 and 7 processes respectively. In Figure 5.2a, the leaders of C_1 and C_2 are the green processes p_2 and p_4 respectively. The Byzantine processes of C_1 and C_2 are the red processes $\{p_3\}$ and $\{p_1, p_2\}$ respectively. We note that in both clusters, the number of Byzantine processes is less than one-third of the size of the cluster: $f < |C|/3$ that is $f_1 = 1 < 4/3$ and $f_2 = 2 < 7/3$.

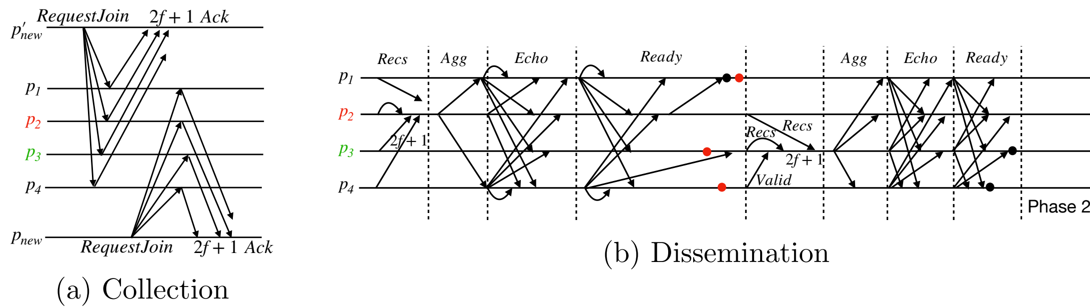
Inter-cluster Broadcast. Figure 5.2a shows an execution of the inter-cluster broadcast protocol. Each cluster has already locally replicated operations; each operation is paired with a certificate of replication which is approval signatures from a quorum of processes in that cluster. The leader of each cluster sends its operations together with their certificates to other clusters as inter-cluster messages *Inter*. To ensure that at least one correct process in the remote cluster receives the message, the leader sends the message to $f + 1$ processes in the remote cluster. In our heterogeneous clusters example, the leader p_2 of C_1 sends the message to $2 + 1 = 3$ processes in C_2, and the leader p_4 of C_2 sends the message to $1 + 1 = 2$ processes in C_1. In the remote cluster, the correct process that receives the *Inter* message then broadcasts the operations as *Local* messages to processes in its own cluster. Thus, if the leaders are correct, all correct processes eventually receive operations from all clusters.

Clustered replication reduces the number of rounds and message complexity for global communication. We just considered the inter-cluster broadcast of phase 2 above. Let us compare the complexity of classical monolithic replication such as PBFT with clustered replication. Consider $n_1 = |\mathcal{C}_1|$, $n_2 = |\mathcal{C}_2|$, and the total number $n = n_1 + n_2$ processes. To

process a single transaction, monolithic replication requires 2 global rounds with message complexity $\mathcal{O}((n_1 + n_2)^2)$. To process 2 transactions in parallel in C_1 and C_2, clustered replication executes phase 1 with 2 local rounds with message complexity $\mathcal{O}(n_1^2 + n_2^2)$, and then phase 2 with 1 global round with message complexity $(f_1+1)+(f_2+1) = \mathcal{O}(n_1+n_2)$, and finally, 1 local round with message complexity $(f_1+1)\times n_1+(f_2+1)\times n_2 = \mathcal{O}(n_1^2+n_2^2)$. Therefore, global communication is reduced from 2 rounds of complexity $\mathcal{O}((n_1 + n_2)^2)$ to 1 round of complexity $\mathcal{O}(n_1 + n_2)$.

Remote Leader Change. A Byzantine leader may behave locally, but skip sending *Inter* messages to other clusters. Let us now consider how the processes of a cluster can change the leader of a remote cluster if they do not receive the expected message from it. Figure 5.2b shows an execution of the remote leader change protocol. The current leader p_2 of the cluster C_1 is Byzantine, and will be changed to the correct process p_3. In cluster C_2, the processes p_3, p_4 and p_5 have not received the operations of C_1, and their timers expire; thus, they broadcast a local complaint *LComplaint* in C_2 about C_1. The processes p_6 and p_7 in C_2 have not already complained, but receive $f_2 + 1 = 3$ complaints from the three processes above. Since at least 1 out of 3 is from a correct process, they amplify the complaint by broadcasting an *LComplaint* message locally. A process accepts the local complain only when it receive it from $2 \times f_2 + 1 = 5$ processes. It can be shown that this prevents a coalition of Byzantine processes from forcing a leader change, and ensures that all local correct processes eventually deliver the complaint. When the first $f_2 + 1 = 3$ processes accept the local complaint, they send a remote complaint *RComplaint*. To make sure that the message is sent to the remote cluster, it is sent by $f_2 + 1$ processes which

contain at least one correct process. In the first three processes, p_3 is correct and sends the remote complaint. The complaint should reach at least one correct process in C_1; thus, p_3 sends it to $f_1 + 1 = 2$ processes in C_1. The process p_1 in C_1 is correct, and receives the remote complaint. It accepts the complain if it carries $2 \times f_2 + 1$ signatures from C_2. It then broadcasts a *Complaint* message locally in C_1. When the correct processes receive the local complaint (at green circles), they move to the next leader p_3. The protocol should deal with complaint replay attacks, and multiple simultaneous change requests, that we will describe in the next section.



(a) Collection

(b) Dissemination

Figure 5.3: Phase 1: Reconfiguration

Reconfiguration. Let's now consider reconfiguration. Reconfiguration not only allows processes to freely join and leave but also supports rebalancing the system to maintain the proximity of processes in a cluster, and similarity of performance across clusters.

Attacks. The reconfiguration requests should be uniformly propagated across clusters, *i.e.*, the configurations that every pair of correct processes (possibly from different clusters) execute in a round should be the same. When they are not, the following Byzantine attacks may arise. Consider two clusters C_1 and C_2 with 4 and 7 processes, and the failure

thresholds $f_1 = 1$ and $f_2 = 2$ respectively. Assume that 3 new processes join C_1 and one of them is Byzantine. The updated C_1 now has 7 processes, and the failure threshold is $f'_1 = 2$. However, assume that the correct processes in C_2 are unaware of the newly joined processes in C_1; they keep the stale failure threshold $f_1 = 1$, and will accept any operations with $2 \times f_1 + 1 = 3$ signatures. If C_1 has a Byzantine leader, it can forge a certificate for a set of operations ops_1: it can get a signature from only one correct process for ops_1. Then, it can also have signatures from itself and the other Byzantine process, to have a total of 3 signatures. It can then make the processes in C_2 accept ops_1 with the forged certificate. However, it can lead the correct processes in C_1 to eventually replicate a different set of operations. Thus, the correct processes in C_1 and C_2 diverge.

Let us now consider another attack in the same setting. Since the correct leader of C_2 has a stale failure threshold $f_1 = 1$, it sends $f_1 + 1 = 2$ inter-cluster broadcast messages to C_1. The receiver processes in C_1 can be both Byzantine, and may drop the message. Then, the timers of the correct processes in C_1 will eventually trigger, and they complain about the leader of C_2. The remote leader change eventually replaces the correct leader in C_2. Unfortunately, the Byzantine processes in C_1 can repeat changing the leader until a Byzantine processes is in control in C_2.

Let's consider the reconfiguration protocol. Processes can request join and leave reconfigurations in phase 1. Clusters communicate only in phase 2. Thus, if the reconfigurations requested in a cluster in phase 1 are processed as they are requested, remote clusters will have an inconsistent view of membership for the local cluster. We explained above that these inconsistencies are unsafe. Therefore, the reconfigurations requested in a round

are locally collected and disseminated in phase 1, are remotely communicated in phase 2, and applied in phase 3 to uniformly update membership for the next round. Thus, in each round, they can be collected as a set, and the order that they are processed in is immaterial. Therefore, collecting them can be taken off the critical path that orders transactions. Thus, as Figure 5.1 shows, reconfigurations are collected and disseminated as a separate workflow in parallel to transaction processing. Figure 5.3 shows example executions for both parts of the reconfiguration protocol, collection and dissemination, which we will describe next.

Collection. In Figure 5.3a, two processes p_{new} and p'_{new} request to join the cluster. Each broadcasts a *RequestJoin* message. When a correct process delivers a *RequestJoin* message, it adds the join request to its set of reconfiguration requests, and responds back by a *Ack* message. A joining process periodically keeps sending *RequestJoin* messages until it receives the *Ack* message with the same configuration from a quorum of $2 \times f + 1 = 3$ processes. It stops then as it learns that Byzantine processes cannot censor the request.

Dissemination. The reconfigurations should be uniformly disseminated to all correct processes in the cluster. Otherwise, as we discussed in the introduction, an inconsistent view of members can lead to accepting fake, or discarding genuine messages. We describe an execution where the leader is Byzantine and is changed; nonetheless, the same set of reconfigurations are uniformly delivered to all well-behaved processes.

As Figure 5.3b shows, later in the first phase, each correct process sends the set of reconfiguration requests that it has collected as *Recs* messages to the leader process p_2. When the leader p_2 receives messages from a quorum, it aggregates the received sets of reconfigurations, and the accompanying signatures, and then starts disseminating them.

Since there is a correct process in the intersection of every pair of quorums, the leader does not miss the requests. In Figure 5.3a and 5.3b, the quorum $\{p_1, p_2, p_3\}$ that p'_{new} receives the state from, and the quorum $\{p_2, p_3, p_4\}$ that the leader p_2 receives requests from intersect in the correct process p_3.

The leader broadcasts the aggregation of the reconfiguration requests that it collected. Upon delivery from the leader, a correct process checks whether the received reconfigurations are valid: they should be accompanied by at least a quorum of signatures for *Recs* messages. As we saw in the collection part, a requesting process makes a quorum of processes store the reconfiguration request. Therefore, the leader cannot drop requested reconfigurations: if the leader drops a request, and hence, any signature from the quorum of processes that stored it, then the remaining processes will be smaller than a quorum, and the leader cannot collect a quorum of signatures. In Figure 5.3b, although the leader p_2 is Byzantine, it has to send the complete aggregated set. However, it only sends it to a subset of processes $\{p_1, p_4\}$. The correct processes p_1 and p_4 that receive a message from the leader echo it. The Byzantine process p_2 echos to them but not p_3. Thus, p_1 and p_4 receive a quorum of 3 *Echo* messages, and broadcast a *Ready* message. The Byzantine process p_2 sends a *Ready* message to only p_1. Thus, only p_1 receives a quorum of 3 *Ready* messages, and delivers the reconfigurations (at the black circle).

The correct processes p_3 and p_4 don't receive enough *Ready* messages, eventually complain about the leader p_2, and change the leader to the correct process p_3 (at the red circles). To preserve uniformity, the new leader p_3 should retrieve the set of reconfigurations that p_1 previously delivered. We will describe later in section 5.4, how the leader retrieves

that set, and makes the remaining correct processes p_3 and p_4 eventually deliver the same set (at black circles).

We note that the classical Byzantine reliable broadcast (BRB) and Byzantine consensus would be inadequate for reconfiguration dissemination. First, in contrast to BRB that guarantees termination only when the sender is correct, the reconfigurations are expected to be eventually delivered in each round even if the initial leader is Byzantine. Thus, to ensure termination, the leader might be changed during dissemination. The challenge is to keep uniformity across leaders. Further, in contrast to BRB where a message from a designated sender is broadcast, and in contrast to consensus where a proposal from one process is decided, this protocol should aggregate and broadcast a collection of reconfigurations from a quorum of processes.

In this section, we saw an overview of the phases (Figure 5.1 and the accompanying description). Next, we first consider the two more important sub-protocols: inter-cluster communication (section 5.3), and reconfiguration (section 5.4). We then revisit the phases, and their sub-protocols (section 5.5).

5.3 Inter-cluster Communication

We will now present the inter-cluster broadcast protocol that propagates operations between clusters, and the heterogeneous remote leader change protocol that detects and changes Byzantine leaders for remote clusters.

State. Each process keeps the set of processes C_j for each cluster. (We use the index $\boldsymbol{i}$ only for the current cluster, and the index j for clusters in general.) The set

C_i keeps track of membership within the current cluster $\boldsymbol{i}$, and is used for intra-cluster communication. The sets C_j that keep track of the members of remote clusters j are used for inter-cluster broadcast. Accordingly, a process has the failure threshold f_j for each cluster C_j as one-third of the size of C_j. Each process also keeps the current round r. Further, it stores the operations $\mathit{operations}_j$ that it receives from each cluster C_j. Each process keeps a set of certificates *certs* for its local operations $\mathit{operations}_{\boldsymbol{i}}$. A certificate for an operation contains at least $2 \times f_{\boldsymbol{i}} + 1$ signatures, and is sent to other clusters together with the operation. The protocol uses authenticated perfect links *apl*, and authenticated best-effort broadcast *abeb* (that were described in section 5.2). Each message m^σ delivered from an authenticated link comes with a signature σ of the sender. (We elide the signature when it is not needed in a context.)

Inter-cluster Broadcast. At the end of phase 1, the local ordering phase, the leader calls the function *inter-broadcast* (Alg. 18) to start the second phase. Each cluster broadcasts its locally ordered operations to remote clusters. As Figure 5.2a shows, this function sends out the batch of operations *ops* of the local cluster together with their certificates *certs* as *Inter* messages to other clusters (at line 11-14). For each remote cluster j, the *Inter* messages are sent to $f_j + 1$ distinct processes. Therefore, at least one correct process at cluster C_j eventually receives the *Inter* message (at line 15). It checks that the certificates are valid: a certificate for an operation from cluster $C_{j'}$ is valid if it contains at least $2 \times f_{j'} + 1$ signatures from the cluster $C_{j'}$. The receiving process then broadcasts the operations as *Local* messages to other processes in its own cluster (at line 16). Upon receiving a *Local* message containing operations *ops* from a remote cluster j' with valid

certificates (at line 17), the process maps j' to *ops* in its *operations* map. It also stops a *timer* that watches the leader of cluster j. (We will consider remote leader change in the next paragraph). When operations from all clusters are received, the process calls the function *execute* to enter phase 3, the ordering and execution phase (at line 21).

Algorithm 18: Inter-cluster Broadcast

```
1  vars:
2    C_j : Set[P]                                  ▷ Processes of each cluster C_j
3    i                                             ▷ The number of the current cluster
4    f_j : ℕ                                       ▷ Failure threshold for C_j
5    r                                             ▷ The current round
6    operations_j ← ∅                              ▷ Operations from each cluster C_j
7    certs                                         ▷ Certificates for operations_i of C_i
8  Uses:
9    apl : AuthenticatedPoint2PointLink
10   abeb : AuthenticatedBestEffortBroadcast in C_i
11 function inter-broadcast(r, ops, certs)
12     foreach C_j, j ≠ i
13         foreach p ∈ P where P ⊆ C_j ∧ |P| = f_j + 1
14             apl request send(p, Inter(r, i, ops, certs))
15 upon apl response deliver(p, Inter(r', j, ops, Σ)) where r' = r ∧ Σ is valid (i.e., Σ
   has at least 2 × f_j + 1 signatures from C_j for each op ∈ ops)
16     abeb request broadcast(Local(r, j, ops, Σ))
17 upon abeb response deliver(p, Local(r', j, ops, Σ)) where r' = r ∧ Σ is valid
18     operations_j ← ops
19     stop timer_j
20     if |dom(operations)| = N    ▷ N is # of clusters then
21         call execute(operations)
```

Heterogeneous Remote Leader Change. Each process waits until it receives operations from other clusters. Therefore, if the leader of a cluster is Byzantine, and avoids sending operations to other clusters, it can stall progress. Consider a system where cluster C_j has a Byzantine leader l. For example in Figure 5.2b, the leader p_2 of C_1 is Byzantine. It acts as a correct leader internally in C_j for the local ordering phase. The correct processes in C_j cannot identify l as a Byzantine leader to replace it. However, l does not follow the

protocol to send its operations to a remote cluster $C_{j'}$. Thus, processes of the cluster $C_{j'}$ cannot proceed to the ordering and execution phase.

Intuition. Let us briefly describe how the local cluster can trigger leader change in a remote cluster. Each process keeps a timer $timer_j$ for the leader of each cluster C_j. It resets the timers for all clusters at the beginning of each round. When a local process does not receive the operations of a remote cluster, and the timer expires, it broadcasts a complaint in its local cluster. When enough local processes complain, the complaint is eventually accepted locally. A subset of local processes that accept a complaint send complaints to remote processes which in turn broadcast it in the remote cluster. Once remote processes receive the remote complaint, they change the remote leader.

A remote process accepts a remote complaint only if it comes with a quorum of signatures from the complaining cluster. This prevents any coalition of Byzantine processes in the complaining cluster to force a remote leader change. However, a Byzantine process in the remote cluster can keep a valid complaint and its accompanying signatures, and launch a replay attack: it can resend the valid complaint to repeatedly change the leader. To prevent this attack, the complaining cluster maintains a complaint number cn_j for each remote cluster C_j, which is incremented on every remote complaint sent to C_j. A remote process maintains the number of complaints received $rcn_{j'}$ from each other cluster $C_{j'}$, and only accepts a complaint with the next expected number, and then increments the number. Therefore, a remote process accepts each remote complaint only once.

Protocol. As Alg. 19 presents, if a local process finds that the timer $timer_j$ for a remote cluster C_j is expired (at line 7), it broadcasts a local complaint *LComplaint* message

Algorithm 19: Heterogeneous Remote Leader Change 1/2

```
1  vars:
2    self                                    ▷ The current process
3    timer_j ← Δ                             ▷ A timer for each cluster C_j
4    cn_j ← rcn_j ← 0                        ▷ # of complaints sent to & received from C_j
5    cs_j ← ∅                                ▷ Complaint signatures for each cluster C_j
6    complained_j ← false                    ▷ If complained about each cluster C_j
7  upon timer_j for remote C_j expires
8  |  abeb request broadcast(LComplaint(j, cn_j, r))
9  |  complained_j ← true
10 upon abeb response deliver(p, LComplaint(j, c, r')^σ) where r' = r ∧
   c = cn_j ∧ operations_j = ⊥
11 |  cs_j ← cs_j ∪ {σ}
12 |  if |cs_j| ≥ f_i + 1 ∧ ¬complained_j then
13 |  |  complained_j ← true
14 |  |  abeb request broadcast(LComplaint(j, c, r))
15 |  if |cs_j| ≥ 2 × f_i + 1 then
16 |  |  let S := first f_i + 1 processes of C_i in
17 |  |  if self ∈ S then
18 |  |  |  apl request send(p, RComplaint(cn_j, i, cs_j, r)), for each p ∈ S' in a set
   |  |  |    S' such that S' ⊆ C_j ∧ |S'| = f_j + 1
19 |  |  cn_j ← cn_j + 1
20 |  |  cs_j ← ∅;   complained_j ← false;   reset timer_j
```

Algorithm 20: Heterogeneous Remote Leader Change 2/2

```
1  upon apl response deliver(p, RComplaint(c, j', Σ, r)) where r = r' ∧ c = rcn_j' ∧ Σ
   contains 2 × f_j' + 1 signatures from C_j'
2  |  abeb request broadcast(Complaint(c, j', Σ))
3  upon abeb response deliver(p, Complaint(c, j', Σ)) where c = rcn_j' ∧ Σ contains
   2 × f_j' + 1 signatures from C_j'
4  |  rcn_j' ← rcn_j' + 1
5  |  if Δ − timer_i > ϵ then
6  |  |  le request next-leader
```

about C_j to processes in its own local cluster (at line 8). In Figure 5.2b, the processes $\{p_3, p_4, p_5\}$ in C_2 send *LComplaint* messages. The message includes the current complaint number cn_j. Once a local process receives a local complaint for a remote cluster C_j with the expected complaint number cn_j, and it has not received operations from that cluster (at line 10), it records the accompanying signature σ in the set of complaint signatures cs_j (at line 11). If the process receives $f_i + 1$ complaint signatures, since at least one is from a correct process, the process amplifies the complaint locally if it has not already complained (at line 12-14). In Figure 5.2b, the processes $\{p_6, p_7\}$ in C_2 amplify the *LComplaint* message.

Once a process receives $2 \times f_i + 1$ complaint signatures (at line 15), it accepts the local complaint. Since there is at least one correct process in the senders, Byzantine processes cannot force a leader change. Further, since the complaint is received from $2 \times f_i + 1$ processes, it can be shown that all correct processes in the local cluster eventually deliver the complaint. The complaint should reach at least one correct process in the remote cluster C_j. Therefore, the remote complaint message *RComplaint* should be sent to at least $f_j + 1$ remote processes.

Further, at least one *correct* process should send these messages. Therefore, at least $f_i + 1$ processes should send it. The first $f_i + 1$ processes of the local cluster (by a predefined order) send the complaint (at line 16); we call them the sender set. In Figure 5.2b, the sender set is $\{p_1, p_2, p_3\}$. The two processes p_1 and p_2 are Byzantine but p_3 is correct and sends the message. If the current process is in the sender set, it sends a remote complaint *RComplaint* message to a subset of C_j of size $f_j + 1$ (at line 17-18). The remote complaint message includes the complaint number cn_j and the collected signatures cs_j. Finally, the

local process increments the complaint number, and resets the state for the next complaint (at line 19-20).

Once a process receives the remote complaint message (at line 1), if the message has the next expected complaint number $rcn_{j'}$, and it carries $2 \times f_{j'} + 1$ signatures from the complaining cluster $C_{j'}$, it broadcasts a *Complaint* message in its own cluster (at line 2). When a process receives the complaint message from its local cluster (at line 3), it performs similar checks to accept it. It then increments the received complaint number $rcn_{j'}$ for the complaining cluster $C_{j'}$, and unless the leader is recently changed, it requests the local leader election module *le* to move to the next leader (at line 4-6). (We will consider the local leader election module *le* in section 5.5.) If the leader is changed recently (*i.e.*, only a small amount of time ϵ is passed since the $timer_i$ is reset to Δ), the protocol avoids requesting to change the leader again so that the new leader is not disrupted. In particular, this happens when multiple remote clusters complain about the same leader at almost the same time.

5.4 Reconfiguration

A process p can issue a *join* or *leave* request to join or leave. Later, it receives a *joined* or *left* response (when the reconfiguration is executed in phase 3). As we showed in Figure 5.1 and briefly described in the overview section 5.2, reconfiguration requests are collected, and then disseminated locally in phase 1. We now consider these two steps.

Collection. As Alg. 21 presents, when a client process p receives a *join* request (at line 6), it broadcasts *RequestJoin* messages in the local cluster (at line 7). In Figure 5.3a,

Algorithm 21: Reconfiguration (Collection)

```
1  request : join, leave
2  response : joined, left
3  vars:
4     recs ← ∅                                        ▷ Set of reconfigurations
5     client-timer ← Δ
6  upon request join
7     abeb request broadcast(RequestJoin(r))
8  upon request leave
9     abeb request broadcast(RequestLeave(r))
10 upon client-timer expires
11    if requested join then
12       abeb request broadcast(RequestJoin(r))
13    else if requested leave then
14       abeb request broadcast(RequestLeave(r))
15    reset client-timer to a longer period
16 upon abeb response deliver(p, RequestJoin^σ(r')) where r = r'
17    recs ← recs ∪ {join(p)^σ}
18    apl request send(p, Ack(C_i, r))
19 upon abeb response deliver(p, RequestLeave^σ(r')) where r = r'
20    recs ← recs ∪ {leave(p)^σ}
21    apl request send(p, Ack(C_i, r))
22 upon apl response deliver(p̄, Ack(C', r')) where |{p̄}| ≥ 2 × f_i + 1 where r = r'
23    stop client-timer
```

Algorithm 22: Reconfiguration (Dissemination)

```
1  Uses:
2     brd : ByzantineReliableDissemination in C_i
3  function send-recs
         ▷ Called by each process before the end of phase 1.
4     brd request broadcast(Recs(r, recs))
5  upon brd response deliver(overline{Recs(r', recs), Σ})^{Σ'} where r' = r ∧ Σ and Σ' are valid.
6     append Reconfig(∪ overline{recs}) to operations_i
7     add Σ, Σ' to certs
8  upon brd response complain(p)
9     call complain(p)
```

two processes p_{new} and p'_{new} request to join. Similarly, when a correct process p receives a *leave* request, it sends out *RequestLeave* messages. The client uses the *client-timer* to track progress while it waits for a response. If the timer expires (at line 10), it resends the messages, and resets the timer to a larger period. When a correct process delivers the *RequestJoin* message from p (at line 16), it adds the reconfiguration request $join(p)$ to its set of collected reconfigurations *recs*, and sends back an *Ack* message (at line 17-18). The steps are similar for the *RequestLeave*. When the requesting process receives *Ack* messages with the same cluster members, and round from a quorum (at line 22), it learns that the request cannot be censored by Byzantine processes; therefore, it stops the timer. In Figure 5.3a, the two joining processes stop the timer when they receive *Ack* from 3 processes.

Dissemination. Before completing the first phase, a correct process calls *send-recs* (Alg. 22 at line 3) that sends a *Recs* message containing the set of reconfiguration requests *recs* that it has collected to the Byzantine Reliable Dissemination (BRD) module (at line 4).

BRD collects messages and disseminates them. It eventually issues a response with a set of collected reconfigurations $\overline{recs}$ (at line 5). The delivery is accompanied by two certificates. The certificate Σ attests that $\overline{recs}$ are collected from at least a quorum of processes. In the collection part, a reconfiguration request was stored in at least a quorum of processes. If Σ is valid, then BRD has collected reconfigurations from at least a quorum of processes. Since there is a correct process in the intersection of two quorums, a Byzantine leader cannot censor the reconfiguration request. The certificate Σ' attests that a quorum of processes voted to deliver the set; therefore, correct processes will eventually deliver the

same set. If the certificates are valid, the receiving process appends the union of $\overline{recs}$ to $operations_i$, and the certificates to *certs* (at line 5-7). The BRD module may complain if the leader does not lead delivery in a timely manner (at line 8-9). The complaint is forwarded to the local leader election module *le*.

Byzantine Reliable Dissemination. In this section, we present the Byzantine Reliable Dissemination (BRD) protocol that we just used. We present it as a general reusable module, that is of independent interest.

Module. BRD accepts a $broadcast(m)$ request from each process. It then collects and disseminates messages m. It issues a response $deliver(M, \Sigma)^{\Sigma'}$ where M is a set of messages, and Σ and Σ' are two sets of signatures. The certificate Σ attests that M is a set of messages from a quorum of processes, and the certificate Σ' attests that M is the only delivered set, and every correct process will eventually deliver it. In our reconfiguration protocol, these certificate are sent to other clusters as a proof of these properties for the dissemination in the current cluster. Further, the component may issue a $complain(p)$ event to complain about the current leader p, and accepts a $new\text{-}leader(p, ts)$ request to set a new leader p with a timestamp ts. Leaders are assumed to have monotonically increasing timestamps. BRD guarantees the following properties. Integrity: A correct process may only deliver messages from at least a quorum of processes. No duplication: Every correct process delivers at most one set of messages. Uniformity: No two correct processes deliver different set of messages. Termination: If all correct processes broadcast messages, then every correct process eventually delivers a set of messages. Totality: If a correct process delivers a set of messages, then all correct processes deliver a set of messages. Validity: If

a correct process delivers a set of messages containing m from a correct sender p, then m was broadcast by p.

Protocol. As Alg. 23 presents, when a process broadcasts a message (at line 13), it stores it and sends it to the leader (at line 14-15). It also resets the timer to watch the leader (at line 16). The leader adds messages and the accompanying signatures that it receives (at line 17) to the set of messages M and signatures Σ (at line 18-20). Once it collects messages from a quorum (at line 21), it broadcasts an aggregation message *Agg* containing M and Σ (at line 22). Massages carry the timestamp *ts* of the current *leader* as well; any message with a stale timestamp is ignored. Upon delivery of the aggregation (at line 1), a correct process accepts it if M is attested by accompanying signatures Σ. The signatures Σ attest M if they include at least a quorum of signatures for the messages M. The signatures serve as a proof that the *leader* has genuinely collected messages from at least a quorum. Therefore, the leader cannot drop the reconfiguration request of a process that has reached out to at least a quorum. For example in Figure 5.3a and 5.3b, the quorum that p'_{new} stored the request at, and the quorum that the leader p_2 receives requests from intersect in the correct process p_3. Even though the leader p_2 is Byzantine, and sends the aggregated set to only a subset of processes $\{p_1, p_4\}$, it cannot drop reconfigurations from the aggregated set.

If the accepting process hasn't sent the *Echo* message, it records (in the variable *echoed*) that it is sending it, and broadcasts the *Echo* message (at line 2-3). In Figure 5.3b, the correct processes p_1 and p_4 that receive an attested set of messages from the leader echo it. Upon delivery of an *Echo* message from a quorum, if the receiving process has not sent

Ready messages (at line 4), it records (in the variable *readied*) that it is sending it, and then broadcasts a *Ready* message (at line 5-6). In Figure 5.3b, processes p_1 and p_4 receive a quorum of 3 *Echo* messages, and broadcast *Ready*.

Algorithm 23: Byzantine Reliable Dissemination (1/3)

```
1  request : broadcast(m), new-leader(p, ts)
2  response : deliver({m̄}, Σ), complain(p)
3  Uses:
4     apl : AuthenticatedPoint2PointLink
5     abeb : AuthenticatedBestEffortBroadcast
6  vars:
7     (leader, ts) ← (p0, 0)
8     my-m ← ⊥
9     echoed, readied, delivered ← false                    ▷ Tracking reliable delivery
10    valid, high-valid ← ⊥                                  ▷ Validated set of requests
11    q, M, Σ ← ∅                     ▷ Collected senders, messages, and signatures
12    timer ← Δ
13 upon request broadcast(m)
14    my-m ← m
15    apl request send(leader, ⟨m, ts⟩)
16    reset timer
17 upon apl response deliver(p, ⟨m, t⟩^σ) where self = leader ∧ t = ts
18    q ← q ∪ {p}
19    M ← M ∪ {m}
20    Σ ← Σ ∪ {σ}
21 upon |q| ≥ 2 × f + 1 ∧ high-valid = ⊥
22    abeb request broadcast(Agg(M, Σ, ts))
```

If the leader changes during the broadcast, some correct processes might have delivered the aggregated messages while others may have not. Thus, to preserve the uniformity of delivered messages across processes, the new leader should retrieve the previously delivered messages, and rebroadcast them. Thus, when a process accepts a sufficiently echoed set, it stores it together with its accompanying signatures, as *valid* (at line 7), and later forwards it to a new leader. In Figure 5.3b, p_1 and p_4 record a *valid* set at the end of the *Echo* step.

Algorithm 24: Byzantine Reliable Dissemination (2/3)

```
1  upon abeb response deliver(p, Agg(M, Σ, t)) where p = leader ∧ t = ts ∧ ¬echoed ∧
   Σ attests M (i.e., Σ has either at least 2 × f + 1 signatures for M, at least 2 × f + 1
   Echo(M) messages, or f + 1 Ready(M) messages)
2      echoed ← true
3      abeb request broadcast(Echo(M, ts))
4  upon abeb response deliver(p̄, Echo(M, t)^σ̄) where |{p̄}| ≥ 2 × f + 1 ∧ t = ts ∧
   ¬readied
5      readied ← true
6      abeb request broadcast(Ready(M, ts))
7      valid ← ⟨M, σ̄, ts⟩
8  upon abeb response deliver(p̄, Ready(M, t)^σ̄) where |{p̄}| ≥ f + 1 ∧ t = ts ∧
   ¬readied
9      readied ← true
10     abeb request broadcast(Ready(M, ts))
11     valid ← ⟨M, σ̄, ts⟩
12 upon abeb response deliver(p̄, Ready(M, t)^σ̄) where |{p̄}| ≥ 2 × f + 1 ∧ t = ts ∧
   ¬delivered
13     delivered ← true
14     response deliver(M, Σ)^σ̄
15     stop timer
```

Algorithm 25: Byzantine Reliable Dissemination (3/3)

```
1  upon timer expires
2      response complain(leader)
3  upon request new-leader(p, t)
4      (leader, ts) ← (p, t)
5      echoed, readied ← false
6      valid, high-valid ← ⊥
7      q, M, Σ ← ∅
8      reset timer
9      if valid ≠ ⊥ then
10         apl request send(leader, Valid(valid))
11     else
12         if my-m ≠ ⊥ then
13             apl request send(leader, ⟨my-m, ts⟩)
14 upon apl response deliver(p, Valid(M, Σ, t)) where self = leader ∧ Σ attests M
   (i.e., Σ has at least 2 × f + 1 Echo(M) messages or f + 1 Ready(M) messages)
15     let ⟨_, _, ht⟩ := high-valid in
16     if t > ht then high-valid ← ⟨M, Σ, t⟩
17     q ← q ∪ {p}
18 upon |q| ≥ 2 × f + 1 ∧ high-valid ≠ ⊥
19     let ⟨M, Σ, _⟩ := high-valid in
20     abeb request broadcast(Agg(M, Σ, ts))
```

When a process receives at least $f + 1$ *Ready* messages (at line 8), at least one of them is correct and has received at least a quorum of *Echo* messages. Therefore, the process trusts the *Ready* message and amplified it: it records that it is sending it, and broadcasts a *Ready* message (at line 9-10). It also records the received messages M and signatures of the received *Ready* messages as *valid* (at line 11), and later forwards it to a new leader.

Finally, when a process receives a quorum of *Ready* messages, and it has not delivered the aggregated messages yet (at line 12), it records (in the variable *delivered*) that it is delivering, delivers the aggregated messages M, and stops the timer (at line 13-15). If a process does not deliver the aggregated messages before the timer times out, it complains about the current leader (at line 1-2). In Figure 5.3b, the correct process p_1 receives a quorum of 3 *Ready* messages, and delivers the reconfigurations (at the black circle). However, the other correct processes do not receive enough *Ready* messages, complain about the leader, and eventually change the leader to the correct process p_3 (at the red circles).

To preserve uniformity, the new leader should retrieve the set of reconfigurations that have been previously delivered. When a process is informed of a new leader (at line 3), it records the new leader and timestamp, resets the state and the timer (at line 4-8), and then sends a message to the new leader to inform him about the current state of dissemination.

If a *valid* set of messages is recorded during the execution with the previous leaders, the process sends it to the new leader (at line 10). Otherwise, it sends the message that it originally broadcast (line 13-15) to the current leader (at line 13). In Figure 5.3b, the two processes p_2 and p_3 send to the new leader the set of reconfigurations that they had collected and sent to the previous leader. However, process p_4 has a *valid* set of reconfigurations.

Let l be the latest leader with the timestamp ts that has guided the system to delivery of a set M at a correct process. Consider the next leader l' with the timestamp ts'. To preserve uniformity, l' should adopt M. In order to find M, l' waits to receive messages from a quorum of processes, and then picks the *valid* set with the largest timestamp. Let us explain why. The set M was delivered only after a quorum of *Ready* messages was received. At least $f+1$ of the senders are correct. A correct process sends a *Ready* message only after receiving $2 \times f + 1$ *Echo* messages, or $f + 1$ *Ready* messages. In both of those cases, the receiving process stores M with ts as *valid*. Thus, at least $f + 1$ correct processes P have stored M with ts as *valid*. Therefore, if l' receives messages from a quorum $(2 \times f + 1)$ of processes, and retrieves any *valid* sets, then M with the largest timestamp ts is retrieved from at least one process in P. The leader l' adopts and broadcasts M. Even if it does not lead to any new delivery of M, any *valid* set that is stored under his leadership will have the same set M with now the larger timestamp ts'.

When the leader receives a *valid* set (at line 14), it checks that the accompanying signatures attest its validity: there are at least $2 \times f + 1$ signatures of *Echo* messages, or $f + 1$ signatures of *Ready* messages. The leader keeps the *valid* set with the highest timestamp as *high-valid* (at line 15-16). Finally, when the leader has collected messages from a quorum, if it has received a *valid* set (at line 18), it broadcasts *high-valid* (at line 20). Otherwise, similar to the first leader (at line 21), it broadcasts the aggregated messages. In Figure 5.3b, the new correct leader p_3 waits for 3 messages, adopts the *valid* set that p_4 sends, goes through the *Echo* and *Ready* steps, and makes the remaining correct processes p_3 and p_4 deliver the same set (at black circles).

5.5 Protocol Phases

In the overview section 5.2, we explained the structure and the three phases of the protocol. as shown in Figure 5.1. We presented two sub-protocols in section 5.3 and section 5.4. In this section, we describe the other sub-protocols. (Full protocols are available in the appendix section 5.8.)

Phase 1: Intra-cluster Replication. Phase 1 has two parallel parts: local ordering and reconfiguration. We considered reconfiguration in section 5.4. We now consider local ordering and leader change.

Local ordering. In order to process a transaction t, clients can issue a request $process(t)$ at any process of any cluster and will receive a $return(t, v)$ response later in phase 3. Each cluster uses a total-order broadcast instance *tob* to propagate transactions to its processes in a uniform order. The protocol is parametric for the total-order broadcast that abstracts the classical monolithic Byzantine replication protocols. Upon delivery of a transaction t from the *tob* the process appends t to $operations_i$ that are received from the local cluster C_i The *tob* delivers a transaction t with a commit certificate σ that is the set of signatures of the quorum that voted to commit t. Each process keeps the set of certificates *certs* for the transactions committed in its local cluster In phase 2, the leader sends the transactions together with their certificates to other clusters. The certificates prevent Byzantine leaders from sending forged transactions to remote clusters.

As we described in section 5.4, a process collects a set of reconfiguration requests *recs*, and then calls the function *send-recs* (Alg. 21, line 3) to send them to the leader who aggregates and uniformly replicates them. A process calls this function towards the end of

phase 1, *i.e.*, when a large fraction of the transaction batch is already ordered. This leaves ample time in the beginning of phase 1 to collect reconfiguration requests, and also leaves enough time at the end of phase 1 to disseminate reconfigurations. Finally, at the end of phase 1, when $operations_i$ contains the batch of both transactions and reconfigurations, the leader calls the function *inter-broadcast* to start phase 2: inter-cluster broadcast (Alg. 18, line 11).

Leader Change. A leader orchestrates both the ordering of transactions in the total-order broadcast *tob* component, and the delivery of the reconfigurations in the Byzantine reliable dissemination *brd* component. However, a leader may be Byzantine, and may not properly lead the cluster. When the delivery of transactions or reconfigurations is not timely, *tob* and *brd* complain. The complains are forwarded as requests to the leader election module *le*. Once *le* receives complaints from a quorum of processes, it eventually issues a response *new-leader*(p, ts) at all correct processes to elect a new leader p with the timestamp ts. Further, we saw in remote leader change (Alg. 19) that when a process receives a *Complaint* message with a quorum of signatures from a remote cluster, it issues a a *next-leader* request to *le* (line 6). Then, *le* issues a response *new-leader*(p, ts) at the process to set the next leader. The module *le* guarantees that the leader for each timestamp is uniform across processes, the timestamps are monotonically increasing, and eventually a correct leader is elected. When a process receives a *new-leader*(p, ts) response, it records the new leader and timestamp, and forwards the new leader event to *tob* and *brd* modules as well. Further, the new leader sends operations of the previous in addition to the current round to remote clusters in case they are behind.

Phase 2: Inter-cluster Communication. We considered this phase in section 5.3.

Phase 3: Execution. At the end of phase 2, a process receives the batches of operations from each other cluster, and calls the *execute* function (Alg. 18 at line 21) that performs the last phase: execution. Processes uniformly order the batches of operations: first, they process the transactions, and then join reconfigurations, and finally leave reconfigurations. Further, they use a predefined order of clusters to order transactions. They then apply each transaction and reconfiguration. If a transaction has been issued by the current process, a *return* response is issued. The process adds joining processes, and removes leaving processes from the local record of processes for the requesting cluster, and if the current process is the leaving process, it issues a *left* response. To kick-start a joining process p, the members of its local cluster send their current state to p. When the joining process receives these messages from a quorum, it sets its current state and configuration, and then issues a *joined* response. The process further updates the failure threshold f_j for each cluster j to less than one-third of the new cluster size. Finally, in order to prepare for the next round, the timers and variables are reset and the round number is incremented.

5.5.1 Correctness

We now state the correctness properties of the sub-protocols and then the end-to-end protocol. The proofs are available in the appendix section 5.9.

Remote Leader Change. The remote leader change protocol satisfies the eventual succession, eventual agreement, and putsch (overthrow) resistance properties. We

say that a process trusts a leader p when it receives the event *new-leader*(p, ts) for a timestamp ts.

Lemma 114 (Eventual Succession) *Let ops be the locally replicated operations of a cluster C in a round. Either ops are eventually delivered to all correct processes of every other cluster in that round, or correct processes in C eventually trust a new leader.*

Lemma 115 (Eventual Agreement) *All correct processes in the same cluster eventually trust the same leader.*

Lemma 116 (Putsch Resistance) *A correct process does not trust a new leader unless at least one correct process complains about the previous leader.*

Inter-cluster Broadcast. In each round, each cluster sends its replicated operations to other clusters as *Inter* messages, that are then propagated there as *Local* messages. Using the above properties for remote leader change, we can prove the following properties for inter-cluster broadcast.

Lemma 117 (Termination) *In every round, every correct process eventually receives operations from each other cluster.*

Lemma 118 (Agreement) *In every round, the operations that every pair of correct processes receive from a cluster are the same.*

Reconfiguration. Each correct process keeps a map C from the cluster identifiers to the set of processes in that cluster. In the execution phase of each round, each correct process applies reconfigurations to C for the next round. The reconfiguration protocol satisfies the following properties.

Lemma 119 (Completeness) *If a correct process p requests to join (or leave) cluster i, then every correct process will eventually have a configuration C such that $p \in C$ (or $p \notin C$).*

Lemma 120 (Accuracy) *Consider a correct process p that has a configuration C in a round, and then another configuration C' in a later round. If a correct process $p \in C'_i \setminus C_i$, then p requested to join the cluster i. Similarly, if a correct process $p \in C_i \setminus C'_i$, then p requested to leave the cluster i.*

Lemma 121 (Uniformity) *In every round, the configurations that every pair of correct processes execute are the same.*

This means that the protocol prevents inconsistent and unsafe configurations that we described in the overview section 5.2. These lemmas use the properties of Byzantine reliable dissemination component that we proved in the appendix subsection 5.9.3.

Replicated System. The replicated system satisfied the following end-to-end safety and liveness properties.

Theorem 122 (Validity) *Every operation that a correct process requests is eventually executed by a correct process.*

Theorem 123 (Agreement) *If a correct process executes an operation in a round then every correct process executes that operation in the same round.*

Theorem 124 (Total-order) *For every pair of operations o and o', if a correct process executes only o, or executes o before o', then every correct process executes o' only after o.*

5.6 Related Work

Classical Replication. Since PBFT [116], the first practical Byzantine replication protocol, the followup works [250, 445, 16, 438, 207, 157, 453, 431, 209, 208] improve different aspects of Byzantine consensus not only for partially synchronous but also asynchronous networks. However, these protocols can only work with closed membership: the set of participants is fixed and known to all participants at the outset. In contrast, our paper proposes a clustered replication protocol whose members can be reconfigured at runtime.

Clustered Replication. Clustered replication systems divide processes into small clusters, and perform consensus within local clusters in parallel. Compared to monolithic replication, clustered replication has fewer number of processes in each cluster; therefore, it exhibits improved performance and scalability. Steward [30] implements a replication protocol where processes are partitioned into multiple sites. A leader site is responsible for driving an inter-site coordination protocol similar to Paxos [263], which may become the bottleneck. ResilientDB [212] alleviates the need for a leader site, and enables higher throughput by letting clusters process their own transactions, and then propagate them. However, it is designed for closed homogeneous clusters. Our clustered replication protocol supports heterogeneity and reconfiguration across clusters which allows more flexible and efficient setups.

In ResilientDB, in order to change a remote Byzantine leader in cluster C_1, each correct process in C_2 sends a complaint message to a corresponding process in C_1. When a process in C_1 delivers a complaint message, it broadcasts it to all processes in C_1. Upon delivering $f_1 + 1$ complain messages, a correct process in C_1 detects leader failure, and

invokes the view-change protocol, which involves two rounds of message-passing. In our protocol, since a complaint message carries enough signatures, once a process in C_1 receives a complaint message, it immediately changes the leader without additional message-passing rounds. Further, only $f_2 + 1$ processes in C_2 (the sender set) send out remote complaint messages to $f_1 + 1$ processes in C_1.

Another line of work is sharding-based consensus. Elastico [316] presents a sharding-based consensus protocol for permissionless blockchains. Further, OmniLedger [247] and RapidChain [449] support reconfiguration for sharding-based consensus. In order to process cross-shard transactions, OmniLedger allows clients to run a two-phase commit protocol to atomically commit or abort a transaction. However, the transactions needs to be gossiped across the network. RapidChain proposes an inter-committee routing protocol and a P2P gossiping protocol to reduce communication. OmniLedger and RapidChain are linearly scalable; however, they suffer from replay attacks in cross-shard commit protocols [407]. In contrast, our protocol provides full replication and avoids complications of cross-shard synchronization. However, any node in the system needs to store all the state, and process every transaction.

Group and Open Membership. A group membership service maintains the set of active processes by installing new views. Since accurate membership is as strong as consensus [121, 128], classical [383, 266, 262, 204] group membership and reconfiguration protocols use consensus to reach an agreement on membership and adjust quorums accordingly. There are two categories of membership services: primary partition, and partitionable [128]. In a primary partition service, as processes leave and join, correct processes install

totally ordered views. In contrast, a partitionable service allows partially ordered views. Our protocol batches reconfigurations in rounds, and totally orders batches by the round number.

Recent works present communication and shared memory abstractions on top of group membership. DBRB [205] provides Byzantine reliable broadcast while processes leave and join. Cogsworth [351] provides Byzantine view-synchronization with linear communication complexity. DBQS [29] preserves consistent read and write quorums for shared memory, and further adjusts the cardinality of quorums according to frequency of failures. In contrast, this paper presents reconfiguration protocols that maintain clustered replication systems as processes leave and join.

SmartMerge [233] provides replication, and uses a commutative, associative and idempotent merge function on reconfiguration requests to avoid consensus. It ensures that all processes eventual perform the merge of all the reconfiguration requests. Dyno [158] provides replication and group membership in the primary partition model. Similar to [309], it uses an instance of consensus to order reconfiguration requests. This paper Ava provides replication as well. However, in contrast to SmartMerge and Dyno that consider monolithic replication systems, Ava presents reconfiguration for clustered replication systems, which enables parallelism and scalability. In addition to uniform configurations in each cluster, it needs to keep the configurations uniform across clusters; otherwise, as we explained in the overview section, safety and liveness properties can be violated. Thus, SmartMerge's eventual consistency is insufficient. Therefore, Ava presents Byzantine Reliable Dissemination to uniformly deliver reconfigurations locally in each cluster, and then inter-cluster

communication to uniformly deliver them across clusters in each round. Further, in contrast to Dyno and [383], Ava ensures that processes that request to join in a round receive the latest state and configurations in that round. This guarantee excludes the need to keep a dedicated configuration history or a sync function. Further, in contrast to Dyno that processes reconfigurations as transactions, Ava takes reconfiguration off the transaction processing pipeline, and instead of multiple consensus instances, uses only one instance of Byzantine Reliable Dissemination for all reconfigurations of a cluster in each round. As the experiments showed, this reduces throughput degradation for transactions processing while processes join and leave.

Solida [15], Hybird Consensus [366], Tendermint [94, 33], Casper [100], Algorand [194], RapidChain [449], and OmniLedeger [247] blockchains combine permissionless and permissioned (Byzantine) consensus to provide both efficiency and open membership [55]. They use permissionless consensus, computational puzzles or verifiable random functions to dynamically choose validators for permissioned consensus. Related works further provide reconfiguration for crash fault-tolerant consensus protocols [309, 403], and reconfiguration protocols for random beacons [72, 71]. In contrast, this paper presents a reconfiguration protocol for Byzantine clustered replication.

5.7 Conclusion

This paper presented heterogeneous and reconfigurable clustered replication. It presented a protocol that adapts to different cluster sizes, and allows processes to join and leave clusters efficiently. Further, it stated and proved the safety and liveness properties of

the protocol. We hope that this project further motivates incorporation of open membership into Byzantine replicated systems, and helps blockchains avoid centralization at nodes with high computational power or stake.

5.8 Protocol Phases

In the overview section 5.2 and Figure 5.1, we explained the structure and the three phases of the protocol. We presented two sub-protocols in section 5.3 and section 5.4. In this section, we elaborate the other sub-protocols.

Phase 1: Phase 1 has two parallel parts. We saw the reconfiguration part in section 5.4. We now consider local ordering and leader change.

Local ordering. We consider local replication for transactions.

In order to process a transaction t, clients can issue a request $\mathit{process}(t)$ at any process of any cluster (at line 15), and will later receive a $\mathit{return}(t, v)$ response. Each cluster uses a total-order broadcast instance *tob* to propagate transactions to its processes in a uniform order. In addition to *broadcast* requests and *deliver* responses, the total-order broadcast abstraction can accept $\mathit{new\text{-}leader}(p, ts)$ requests to install the new leader p with the timestamp ts, and can issue $\mathit{complain}(p)$ responses to complain about the leader p. The protocol is parametric for the total-order broadcast. The total-order broadcast abstracts the classical monolithic Byzantine replication protocols. If a complaint is received from *tob*, (at line 25), it is forwarded to the leader election module *le* (in Alg. 27).

Upon receiving a $\mathit{process}(t)$ request (at line 15), the process uses the *tob* to broadcast the transaction in its own cluster (at line 16). Each process stores the operations

operations$_j$ that it receives from each cluster C_j. Upon delivery of a transaction t from the *tob* (at line 17), the process appends t to *operations*$_{\boldsymbol{i}}$ received from this cluster $C_{\boldsymbol{i}}$ (at line 18). Each process keeps the number $\boldsymbol{i}$ of the current cluster $C_{\boldsymbol{i}}$ that it is a member of. (We use the index $\boldsymbol{i}$ only for the current cluster, and the index j for other clusters.) The *tob* delivers a transaction t with a commit certificate σ that is the set of signatures of the quorum that committed t. Each process keeps the set of certificates *certs* for the transactions committed in its local cluster (at line 19). In the next phase, the leader sends the transactions together with their certificates to other clusters. The certificates prevent Byzantine leaders from sending forged transactions.

In parallel to receiving *process*(t) requests and ordering transactions t, processes can receive and propagate *join* and *leave* reconfiguration requests. We will describe the reconfiguration protocol in the next subsection. In each round, the processes of each cluster should agree on the reconfigurations before the end of the intra-cluster replication phase (phase 1). The reconfigurations are then propagated to other clusters in the inter-cluster broadcast phase (phase 2). In phase 1, a process collects the set of reconfiguration requests *recs*. It then calls the function *send-recs* (at line 21) to send the set of reconfigurations it has collected to the leader who aggregates and uniformly replicates them. In section 5.4, we presented the Byzantine Reliable Dissemination component that collects and sends reconfigurations to the leader. A process calls this function towards the end of phase 1, *i.e.*, when a large fraction α of the transaction batch is already ordered (at line 20). This leaves ample time in the beginning of phase 1 to accept reconfiguration requests, and also leaves enough time at the end of phase 1 to reach agreement for the reconfigurations. Finally,

at the end of phase 1, when $operations_i$ contains both the batch of transactions and the reconfigurations (line 22), if the current process (denoted as **self**) is the leader (line 23), it calls the function *inter-broadcast* (at line 24) to start the inter-cluster broadcast phase (phase 2).

Algorithm 26: Local Ordering

```
1  request : process(t)
2  response : return(t, v)
3  Uses:
4    tob : TotalOrderBroadcast
5      request : broadcast(t), new-leader(p, ts)
6      response : deliver(p, t), complain(p)
7  vars:
8    r                                      ▷ The current round
9    i                                      ▷ The number of the current cluster
10   self                                   ▷ The current process
11   leader : P ← p_0^i                     ▷ The leader of current cluster C_i
12   ts ← 0                                 ▷ Timestamp for leader
13   operations_j ← ∅                       ▷ Operations from each cluster C_j
14   certs                                  ▷ Certificates for operations_i of C_i
15 upon request process(t)
16   tob request broadcast(t)
17 upon tob response deliver(p, t^σ)
18   append Trans(p, t) to operations_i
19   add σ to certs
20   if |operations_i| = batch-size × α then
21     call send-recs()
22   else if |operations_i| = batch-size + 1 then
         ▷ batch-size transactions + 1 reconfiguration set
23     if self = leader then
24       inter-broadcast(r, operations_i, certs)
25 upon tob response complain(p)
26   call complain(p)
```

Leader Change. A leader orchestrates both the ordering of transactions in the total-order broadcast, and the delivery of the reconfigurations. However, a leader may be Byzantine, and may not properly lead the cluster. Therefore, as presented in Alg. 27, the protocol monitors and changes leaders. As we described, the total-order broadcast *tob* (Alg. 26 at line 26) and the Byzantine reliable dissemination *brd* (Alg. 21 at line 9) complain

when the delivery of transactions or reconfigurations is not timely. The complains are sent to the leader election module *le* (at line 7-8).

The protocol uses the classical leader election module *le*. The implementation of this module is presented in Alg. 29. Once a quorum of processes send complain requests to *le*, it eventually issues a response *new-leader*(p, ts) at all correct processes to elect a new leader p with the timestamp ts. Further, if the current process sends a *next-leader* request to the module, it issues a response *new-leader* at the current process. This module guarantees that the leader for each timestamp is uniform across processes, the timestamps are monotonically increasing, and eventually a correct leader is elected.

When a process receives a *new-leader*(p, ts) response (at line 9), it records the new leader and timestamp (at line 10), and forwards the new leader event to the total-order broadcast *tob* and Byzantine reliable dissemination *brd* modules as well (at line 11-12). Further, the previous leader might have failed to communicate the operations of the previous round to other clusters. As we will describe next, clusters wait for the operations of each other in each round; therefore, a remote cluster can fall behind by at most one round. Thus, the new leader sends operations of the previous in addition to the current round (at line 14-18).

Phase 2: We already considered this phase in section 5.3.

Phase 3: Execution. At the end of the inter-cluster communication phase, a process receives the batches of operations from each other cluster. It then calls the *execute* function (Alg. 18 at line 21) that performs the last phase: execution (at Alg. 31). Processes

Algorithm 27: Leader Change

```
1  Uses:
2    le : LeaderElection
3      request : complain(p), next-leader
4      response : new-leader(p, ts)
5  vars:
6    p-ops, p-certs                          ▷ ops and certs of the previous round
7  function complain(p)
8    le request complain(p)
9  upon le response new-leader(p, ts′)
10   ⟨leader, ts⟩ ← ⟨p, ts′⟩
11   tob request new-leader(leader, ts)
12   brd request new-leader(leader, ts)
13   reset timer_i
14   if leader = self then
15     if |operations_i| = batch-size then
16       call inter-broadcast(r, operations_i, certs)
17     if r > 1 then
18       call inter-broadcast(r − 1, p-ops, p-certs)
```

Algorithm 28: Leader Election (1/2)

```
1  Implements: Leader Election
2    request : complain(p)
3    response : new-leader(p, ts)
4    request : next-leader
5  Uses:
6    abeb : AuthenticatedBestEffortBroadcast
7  vars:
8    ts ← 1
9    C ← ∅                                   ▷ Set of complaining processes
10   c ← false                               ▷ Complained
11 upon request complain(p)
12   if ¬c then
13     call send-complain()
14 function send-complain()
15   c ← true
16   abeb request broadcast(Complaint(ts))
17 upon abeb response deliver(p, Complaint(ts′)) where ts = ts′
18   C ← C ∪ {p}
19   if |C| ≥ f + 1 ∧ ¬c then
20     call send-complain()
21   if |C| ≥ 2 × f(i) + 1 then
22     call change()
```

Algorithm 29: Leader Election (2/2)

```
1 function change()
2     ts ← ts + 1
3     C ← ∅
4     c ← false
5     response new-leader(p_{ts mod N}, ts)
                                  ▷ Choose leaders in a round robin order.
                                  ▷ N is the number of processes.
6 upon request next-leader
7     call change()
```

uniformly *order* the batches of operations: first, they process the transactions, and then the reconfigurations, and further, use a predefined order of clusters to order transactions (at line 4). Then, they process each operation: they apply each transaction and reconfiguration (at line 6-13). If a transaction has been issued by the current process, a *return* response is issued (at line 9). Finally, in order to prepare for the next round, the timers and variables are reset and the round number is incremented (at line 15-20).

Application of Reconfigurations. The function *reconfigure* is called for each set of reconfigurations *rc* from a cluster j (at line 1). First, the process adds joining processes, and removes leaving processes from the set of processes C_j of cluster j (at line 5 and 7). Then the function *kickstart* is called on the reconfigurations of the local cluster *irc* (at line 14). The function *kickstart* (at line 1) processes all the joins before the leave reconfigurations. We keep this specific order since leaving processes may still need to send additional messages for the new processes. If they leave first, then the new processes will not be able to collect enough states to start the execution. If the leave is for the current process, it issues a *left* response (at line 15). To kick-start a new process p, the members of its local cluster send a *CurrState* message to p (at line 13). The message contains the local *state*, the current round number r, and the cluster members C. Further, the process resets its *echoed*, *readied*,

delivered, and *valid* variables. When a correct process receives *CurrState* messages with the same state s', cluster members C', and round r' from a quorum (at line 19), the process sets its *state*, cluster C, and round r to the received values. It then issues a *joined* response (at line 21). After an addition or a removal, the process further updates the failure threshold f_j for the cluster j to less than one-third of the new cluster size (at line 8).

Algorithm 30: Phase 3: Execution (1/2)

```
1  vars:
2    state                                                    ▷ Process state
3  function execute(operations)
4    foreach operations_j ∈ order(operations)
5      foreach o ∈ operations_j
6        match o
7          case Trans(p, t)  ⇒
8            ⟨state, v⟩ ← t(state)
9            if p = self then response return(t, v)
10         case Reconfig(rc)  ⇒
11           call reconfigure(j, rc)
12           if j = i then
13             irc ← rc
14   call kickstart(irc)
15   p-ops ← operations_j;   p-certs ← certs
16   foreach cluster C_j
17     reset timer_j
18     operations_j ← ∅;   certs ← ∅
19     cn_j ← rcn_j ← 0
20   r ← r + 1
```

Algorithm 31: Phase 3: Execution (2/2)

```
1  function reconfigure(j, rc)
       ▷ Function reconfigure is called in Phase 3.
2      foreach o ∈ rc
3          match o
4              case join(p) ⇒
5                  C_j ← C_j ∪ {p}
6              case leave(p) ⇒
7                  C_j ← C_j \ {p}
8              f_j = ⌊(|C_j| − 1)/3⌋
9  function kickstart(rc)
10     foreach o ∈ rc        ▷ First joins and then leaves.
11         match o
12             case join(p) ⇒
13                 apl request send(p, CurrState(state, C, r))
14             case leave(p) ⇒
15                 if p = self then response left
16     recs ← recs \ {rc}
17     echoed ← readied ← delivered ← false
18     valid ← ⊥
19 upon apl response deliver(p̄, CurrState(s′, C′, r′)) where |{p̄}| ≥ 2 × f_i + 1
20     state ← s′;   r ← r′;   C ← C′
21     response joined
```

5.9 Proofs

5.9.1 Remote Leader Change

Lemma 114 (Eventual Succession). *Let ops be the locally replicated operations of a cluster C in a round. Either ops are eventually delivered to all correct processes of every other cluster in that round, or correct processes in C eventually trust a new leader.*

Proof. Let C_2 be any other cluster in the system except C. There are two cases regarding the delivery of m in cluster C_2.

In the first case, at least one correct process p in C_2 delivers m. Then, it uses *rb* to broadcasts m to all members of the local cluster at Alg. 18, line 16. By validity of reliable broadcast, all the correct processes in C_2 deliver m.

In the second case, none of the correct processes in C_2 delivers m. We prove that processes in C_2 will invoke a remote leader change for C and finally correct processes in C trust a new leader. If none of the correct processes of C_2 delivers m, then their timers will eventually be triggered at Alg. 19, line 7 and all of the correct processes broadcast *LComplaint* at line 8. Thus, the signatures of all of them are stored in cs_1 variable at line 11. Since there are at least $2 \times f_2 + 1$ correct processes in cluster C_2, all the correct processes eventually receive enough *LComplaint* messages, and cs_1 will be large enough. Thus, $f_2 + 1$ processes in C_2 send *RComplaint* messages, and each send it to $f + 1$ distinct processes in C at line 18. Thus, at least one correct process in C eventually delivers the *RComplaint* message at line 1 and verifies the validity of the accompanying signatures Σ. Then, it broadcasts the *Complaint* message locally at line 2. By validity of *abeb*, all the correct processes in C deliver the complain at line 3, and request the leader election module

to move to the next leader at line 6. Thus, the leader election module will eventually choose a new leader. Thus, all the correct processes in C will eventually trust a new leader at Alg. 27 line 9-10.

■

Lemma 125 (Local Complaint Synchronization) *If a correct process in cluster C_i installs $cn_j = k$, then all the correct processes in C_i eventually install $cn_j = k$.*

Proof. We prove this lemma by induction.

For $cn_j = 0$, all the correct processes assign cn_j to be the same value 0 at initialization.

The induction hypothesis is that if a correct process installs $cn_j = k$, then all the correct processes eventually install $cn_j = k$.

We prove that if a correct process in cluster C_i installs $cn_j = k + 1$, then all the correct processes in C_i eventually install $cn_j = k + 1$

A correct process p increments cn_j to $k + 1$ at line 19 after verifying $2f_i + 1$ *LComplaint* messages has been delivered for the same $cn_j = k$ at line 15. Thus at least $f_i + 1$ correct processes have broadcast *LComplaint* messages for $cn_j = k$. By the validity of *abeb*, all the correct processes eventually delivers at least $f_i + 1$ consistent *LComplaint* messages at line 12 and verify the complaint counter: by induction hypothesis and p installed $cn_j = k$, all the correct processes eventually install $cn_j = k$. Then correct processes amplify the complain by broadcasting *LComplaint* messages for $cn_j = k$ at line 14. There are at least $2f_i+1$ correct processes in cluster C_i. By the validity of *abeb*, eventually at least $2f_i+1$ *LComplaint* messages are delivered to all correct processes at line 15 and they increment

cn_j to $k+1$ at line 19. Therefore, all the correct processes install $cn_j = k+1$.

We conclude the induction proof: for $k \geq 0$, if a correct process install $cn_j = k$, then all the correct processes install $cn_j = k$. ■

Lemma 126 (Remote Complaint Synchronization) *If a correct process in cluster C_i installs $rcn_j = k$, then all the correct processes in C_i eventually install $rcn_j = k$.*

Proof. We prove this lemma by induction.

For $rcn_j = 0$, all the correct processes assign rcn_j to be the same value 0 at initialization.

The induction hypothesis is that if a correct process in cluster C_i installs $rcn_j = k$, then all the correct processes eventually install $rcn_j = k$.

We prove that if a correct process p in cluster C_i installs $rcn_j = k+1$, then all the correct processes in C_i eventually install $rcn_j = k+1$

A correct process p increments rcn_j to $k+1$ at line 4 after verifying $2f_j+1$ *LComplaint* messages was in Σ for the same $rcn_j = k$ at line 15. Thus by Theorem 125 and Σ verifies that a correct process in C_j installed $cn_j = k+1$, all the correct processes in C_j eventually install $cn_j = k+1$ at line 19. There are at most f_j Byzantine processes in cluster C_j and $\mathcal{S}$ contains f_j+1 processes, therefore at least one correct process in $\mathcal{S}$ sends $RComplaint(k, j, \Sigma, r)$ messages to f_i+1 processes in C_i. By the validity of *apl*, at least one correct process in C_i receives the *RComplaint* message at line 1 and broadcasts $Complaint(k, j, \Sigma)$ message at line 2. By the validity of *abeb*, all the correct processes in C_i eventually delivers *Complaint* messages at line 3 and verify the complaint counter: by induction hypothesis and p installed $rcn_j = k$, all the correct processes eventually install

$rcn_j = k$. They increment the remote complaint counter rcn_j to $k+1$ line 4. Therefore, all the correct processes install $rcn_j = k+1$.

We conclude the induction proof: for $k \geq 0$, if a correct process install $rcn_j = k$, then all the correct processes install $rcn_j = k$. ∎

Lemma 115 (Eventual Agreement). All correct processes in the same cluster eventually trust the same leader.

Proof. We prove this lemma in three steps. Firstly, we prove if a correct process in C_i issue response *new-leader* for *ts*, then eventually all correct process in C_i issue response *new-leader* for *ts*. Secondly, we prove that eventually all the correct process stop changing leader and stay in the same timestamp. Finally, since the leader is deterministically chosen according to the timestamp and cluster membership, we prove that eventually all the correct process eventually trust the same leader.

For the first statement, *le* issue response for two type of requests: *complain* and *next-leader*. For *complain* request, we directly use the eventual agreement property of underlying module. For *next-leader* request, a correct process p in cluster C_i requests a *next-leader* at line 6. Let us assume that p installs $rcn_j = n$ before the *next-leader* request at line 4. By Theorem 126, all the correct processes in C_i eventually install $rcn_j = n$. By assumption, this request is apart from the previous remote leader change events and $\Delta - timer_i > \epsilon$. Then all the correct process request the *next-leader* for the same *Complaint* message. Therefore the *ts* at all correct processes are eventually the same.

For the second statement, correct processes eventually wait long enough for a correct leader to complete inter-broadcast phase: the timer for remote leader change increases

exponentially and eventually, all the messages are delivered within a bounded delay after GST. When all *Complaint* messages have been received, all the correct processes in the same cluster do not issue new complains and by Theorem 116, they stay in the same ts.

For the third statement, by Theorem 121, all the correct processes in the same cluster maintain a consistent group membership for each round. Then all of them deterministically choose the same process as leader based on group member and timestamp. ■

Lemma 116 (Putsch resistance). A correct process does not trust a new leader unless at least one correct process complains about the previous leader.

Proof. The correct process requests the leader election module to trust the next leader at Alg. 19, line 6. This request is after receiving a *Complaint* message at line 3 with the following checks: (1) the expected next complaint counter rcn_j is equal to the received complain number c, and (2) the signatures Σ include at least $2 \times f_j + 1$ signatures from C_j. The first check prevents replay attacks; thus, no complaints about previous leaders can be reused. Therefore, all the signatures in Σ are complaints for the current leader. The second one implies that a correct process in C_j sent the *RComplaint* message after receiving $2 \times f_j + 1$ *LComplaint* messages at line 15. Thus, at least $f_j + 1$ correct processes sent *LComplaint* messages. A correct process sends a *LComplaint* message at two places: (1) the timer triggers at line 7; (2) the process amplifies the received complaints at line 12. The first case reached the conclusion. In the second case, a correct process only amplifies after receiving $f_j + 1$ *LComplaint* messages. Thus, at least one correct process sent a *LComplaint* message with the same two cases as above. This second case is the inductive

case, and the first case is the base case. Since the number of processes is finite, by induction, this case is reduced to the first case in a finite number of steps. ■

5.9.2 Inter-cluster Broadcast

Lemma 117 (Inter Broadcast Termination). In every round, every correct process eventually receives operations from each other cluster.

Proof. We prove the termination property for inter-cluster broadcast with the help of Theorem 114. A leader of cluster i should send *Inter* message to $f_j + 1$ processes in cluster j for all $i \neq j$ at line 14. By the validity of remote leader change, either this *Inter* message was delivered to all correct processes in cluster j or all the correct processes in cluster i change a leader. In the first case we conclude the proof. In the second case, eventually the correct processes in cluster i trust a correct leader. The correct leader sends *Inter* messages to $f_j + 1$ processes in cluster j. By the validity of *apl*, at least one correct process p in cluster j delivers the *Inter* message at line 15. Then p broadcasts the received content in *Local* message at line 16. By the validity of *abeb*, all the correct processes in cluster j eventually deliver the *Local* message at line 17. We generalize the same reasoning for all the other cluster and conclude the proof. ■

Lemma 118 (Inter Broadcast Agreement). In every round, the operations that every pair of correct processes receive from a cluster are the same.

Proof. Let process p receives $Local(r, j, ops, \Sigma)$ and p' receives $Local(r, j, ops', \Sigma')$. Correct processes only delivery valid *Local* messages, which means Σ attests *ops* and Σ' attests ops'. Then Σ and Σ' both contains $2f + 1$ commit signatures for each operation in *ops* and ops'. By the agreement property of TOB in the first phase and $|ops| = |ops'|$,

ops and ops' contains the same set of operations. By the total order property of the TOB, operations in ops and ops' have the same order. Thus, $ops = ops'$. ■

5.9.3 Byzantine Reliable Dissemination

Lemma 127 (Integrity) *The delivered set contains at least a quorum of messages from distinct processes.*

Proof. A set of messages is delivered at line 14 which is after the delivery of $2f_i+1$ of *Ready* messages (at line 12). At least f_i+1 correct processes sent *Ready* messages since there are only f_i Byzantine processes in a cluster i. A correct process only sends *Ready* message when it receives $2f_i+1$ *Echo* messages or f_i+1 *Ready* messages. Then by induction, at least $2f_i+1$ *Echo* messages were received by a correct process. Then at least f_i+1 correct processes sent *Echo* messages. A correct process only sends *Echo* messages when it verifies M is valid (at line 1). A M is valid if and only if Σ includes $2f_i+1$ distinct signatures and M is the union of all the m sets in those messages; Or M is adopted from the *valid* and Σ contains $2f_i+1$ *Echo* or f_i+1 *Ready* messages. In the first case, the delivered M contains at least a quorum of m. In the second case, by induction M was in $2f_i+1$ of *Echo* messages and the correct processes who sent the *Echo* message verify that M originally was a union of $2f_i+1$ m. ■

Lemma 128 (Termination) *If all correct processes broadcast messages then every correct process eventually delivers a set of messages.*

Proof. We consider two cases based on whether there is a correct process delivered a set of messages.

Case 1: If there is a correct process that delivers, then eventually all the correct processes deliver. A correct process delivers M after receiving $2f_i + 1$ *Ready* message at line 12. Then at least $f_i + 1$ correct processes broadcast the *Ready* message at line 6. By the validity of *abeb*, eventually all the correct processes deliver $f_i + 1$ *Ready* message at line 8 and broadcast the same message at line 10. Eventually, all the correct processes deliver $2f_i + 1$ *Ready* messages and issue delivery response (at line 14).

Otherwise, Case 2: if no correct process delivers, then each correct process complains about the current leader. Then by the eventual agreement property of the Byzantine leader election, all the correct processes eventually trust the same correct leader. Upon the last leader election delivered at line 3, all the correct processes send *Valid* or *my-m* to the correct leader at line 10 or line 13. Since the set of correct processes is a quorum, then the correct leader either delivers a quorum of *my-m* messages at line 21 or a *Valid* message at line 14. Then we have two cases, either there is a valid *valid* or not. In the first case, the correct leader adopts M from *valid*. In the second case, the correct leader composes a new set of reconfiguration requests. Both cases can be verified and accepted by correct processes at line 1. Then all the correct processes send *Echo* message at line 3 and eventually $2f_i + 1$ *Echo* messages are delivered to all the correct processes. Then all the correct processes send *Ready* message at line 6 and eventually $2f_i + 1$ *Ready* message are delivered to all correct processes. Then all the correct processes issue delivery response at line 14 and we conclude the proof. ∎

Lemma 129 (Uniformity) *No correct processes deliver different set of messages*

Proof. There are two cases regarding the delivery of messages for p_1 and p_2:

either they deliver messages with the same ts or different ts.

In the first case, since any pair of quorums has a correct process in the intersection, if p_1 delivers M_1 and p_2 delivers M_2, $M_1 = M_2$. Otherwise, the correct process sends different *Ready* messages for the same round and ts, which is not permitted by the protocol (at line 6, line 10).

In the second case, let us assume that p_1 delivers first with timestamp ts_1 and then p_2 delivers with another timestamp ts_2. Without losing generality, let us assume that $ts_1 < ts_2$. If p_1 delivers M_1 with ts_1, then p_1 receives at least a quorum of *Ready* messages. A correct process set its *valid* before sending *Ready* messages (at line 7, line 11). Therefore, at least $f_i + 1$ correct processes set their *valid* variable with M_1. For the next timestamp $ts_1 < ts_i le ts_2$, it collects a quorum of *my-m* messages or at least one *Valid* message. By assumption, cluster i has $3f_i + 1$ members in total, then at most $2f_i$ processes have not set *valid* and can send *my-m* message, which is not a quorum. Therefore, the leader for ts_i waits for the *Valid* message and adopts its value. Valid *valid* requires either $2f_i + 1$ *Echo* messages or $f_i + 1$ *Ready* messages for the same ts. By induction, since there are only f_i Byzantine processes, a correct process receives $2f_i + 1$ *Echo* messages before sending out *Ready* messages and triggering the amplification. Since any pair of quorums has a correct process in the intersection, there is only one M that can be echoed by a quorum of processes and appears in *valid*. The leader for ts_i can only propose M_1 that will be accepted by correct processes at line 1. From ts_i to ts_2, the *valid* can only be updated to the same M_1. Then when p_2 delivers M_2 in ts_2, $M_2 = M_1$. ■

Lemma 130 (No duplication) *Every correct process delivers at most one set of messages*

Proof. This lemma follows directly from the condition (at line 12) before the delivery response is issued at line 14. ■

Lemma 131 (Validity) *If a correct process delivers a set of messages containing m from a correct sender p, then m was broadcast by p*

Proof. If a correct process delivers a set of messages, then it receives a quorum of *Ready* messages. A ready message is send by a correct process if it receives a quorum of *Echo* messages or $f+1$ ready messages. Since there are only f Byzantine processes, then by induction, the first ready message sent by a correct process is because of receiving a quorum of echo messages. A correct process only send echo message if delivers the *Agg* from the leader with valid certificate. A valid *Agg* message states that M is either collected from a quorum of distinct processes through *apl* or adopted from the previous leader. For the first case, by the validity of *apl*, if the sender of m is correct, then it sends m to the leader. For the second case, M can be adopted only if it carries a certificate with a quorum of *Echo* messages for M or $f+1$ *Ready* messages for m. By the same induction, the messages contained in M is broadcast by its sender p if p is correct. ■

5.9.4 Reconfiguration

Lemma 119 (Completeness). If a correct process p requests to join (or leave) cluster i, then every correct process will eventually have a configuration C such that $p \in C$ (or $p \notin C$).

Proof. We prove the completeness in two steps: first we prove that all the reconfiguration requests will be in a prepared state which we will formally define later; then we

prove that all the prepared reconfiguration requests will be delivered within one round.

We define that a new process prepares a join request when it receives at least a quorum of replies from the existing replicas. Our protocol guarantees that a new process officially joins the system in the round it is prepared. Similarly, we define a leaving process that prepares a leave request when its *RequestLeave* message has been delivered to a quorum of existing replicas. Our protocol guarantees that a leaving process officially leaves the system in the round it is prepared.

For the first statement, when a correct process p requests to join (or leave) the cluster C_i, it sends out *RequestJoin* (or *RequestLeave*) messages to all the existing processes at line 7 (or at line 9). If p's request is not installed in a long time line 10, it resends the *RequestJoin* (or *RequestLeave*) message and doubles the timer at line 12 (or line 14). Therefore *RequestJoin* (or *RequestLeave*) messages sent out by p at line 7 will be delivered at all the correct processes in C_i in the first phase at line 16 after GST. Upon receiving the *RequestJoin* and *RequestLeave* message at line 17 and line 20, correct processes in the system add the reconfiguration request into their *recs* variable. Since all the correct processes in a cluster is a quorum, p's reconfiguration request is eventually prepared.

We prove the second statement in two steps. First, we prove that any set of installed reconfiguration requests at round r includes p's reconfiguration request. Second, we prove that eventually, all correct processes install a set of reconfiguration requests in round r.

For the first step, at the end of the local ordering phase of each round at line 4, correct processes use Byzantine reliable dissemination module to deliver the reconfiguration

requests *recs* that they have collected. Assume that p's reconfiguration request is prepared in round r. By the integrity of BRD, the delivered set contains a quorum of messages send by distinct processes. Since every pair of quorums have at least one correct process in their intersection, at round r, there is always a correct process which sends p's reconfiguration request in the BRD message and the message is included in the delivered set.

For the second step, we consider the delivery of reconfiguration requests for both local and remote clusters.

For the remote clusters, by Theorem 117 all the correct processes in the remote cluster deliver *Local* message, which is verified to contain reconfiguration requests at line 17. Correct processes eventually receives all the *Local* message at line 20 and install reconfiguration at line 21.

For the local cluster, by the termination property of BRD, all the correct nodes in the local cluster eventually deliver a set of reconfiguration requests through BRD at line 5. They insert the reconfiguration requests at line 6. By Theorem 117, all the correct processes receive enough *Local* message and install the reconfiguration requests at line 1.

In conclusion, a set of reconfiguration requests is eventually installed at all the correct processes and we conclude the second step. ■

Lemma 121 (Uniformity). In every round, the configurations that every pair of correct processes execute are the same.

Proof. Let us assume that two correct processes p_1 and p_2 installed new configurations. The correct process installs new group membership at line 1, which is at the order

and execution phase. We prove agreement for correct processes in both local and remote clusters.

For the local cluster, a correct process installs a reconfiguration request from $operations_i$ at line 11. $operations_i$ is updated at line 6, which is after the delivery of an instance of BRD at line 5. By the uniformity property of BRD, all the correct processes deliver the same set of messages. Since the installation of new membership is deterministic and only dependent on the set of reconfiguration requests, we have $C = C'$.

For remote cluster reconfiguration, a correct process in cluster i installs the reconfiguration requests for cluster j at the order and execution phase at line 1. $operations_j$ is updated after verifying the σ at line 17. σ is valid if and only if for each reconfiguration request in T, it contains a quorum of signatures from cluster j in round r. By Theorem 118, the reconfiguration requests installed at cluster j are the same. Therefore, we conclude $C = C'$.

■

Lemma 120 (Accuracy). Consider a correct process p that has a configuration C in a round, and then another configuration C' in a later round. If a correct process $p \in C'_i \setminus C_i$, then p requested to join the cluster i. Similarly, if a correct process $p \in C_i \setminus C'_i$, then p requested to leave the cluster i.

Proof. Since we have $p_n \in C_2 \setminus C_1 \wedge r_2 > r_1$, p_n is not originally a member of this cluster. The cluster membership is updated at line 5, which is after verifying each reconfiguration request is valid: each reconfiguration request is delivered after a quorum of *Ready* messages. At line 1, every correct process checks the validity of *rc*, including its

signatures from p_n. By the authenticity of *apl*, if p_n is correct, then it is the sender of the *RequestJoin* messages and thus requested to join. The same reasoning applies to *leave* requests.

■

5.9.5 Replication System

By Theorem 121, at the beginning of each round, all the correct processes have the same configuration. Thus during the execution of each round, all the correct processes maintain a static membership and we prove termination and total order properties for each round. We prove validity for eventual progress.

Theorem 122 (Validity). Every operation that a correct process requests is eventually executed by a correct process.

Proof. Based on the validity of the underlying TOB protocol in the first phase, if a valid operation o is submitted to a cluster i (at line 15), then o is eventually delivered at a correct process p in C_i at line 17 and included in $operations_i$ (at line 18). By the Theorem 117, each correct process receives *Local* message from each other cluster and call *execute* function (at line 3). Since o was included in $operations_i$, it is executed at line 8 and we conclude the proof. ■

Theorem 123 (Agreement). If a correct process executes an operation in a round then every correct process executes that operation in the same round.

Proof. A correct process deliver a operation in the execution phase (at Alg. 31), which is stored in *operations*. *operations* are updated in the inter-cluster phase (at Alg. 18) for remote clusters and in the local ordering phase (at line 18) for the local cluster. Based

on whether o is an operation from the local cluster, we prove the termination in two cases.

Case 1: o is from the local cluster. Then we prove that o will be delivered locally and remotely in round r. For the local cluster, we can directly use the termination property provided by the underlying TOB protocol: all the correct processes eventually deliver o. Since correct processes in the local cluster are waiting for a fixed number of operations to be delivered in a batch for each round, they will not move to the next round before they deliver o in round r. Then we proved that o will be delivered locally in round r.

For the remote delivery of o, by the total order and termination property of underlying TOB protocol, o will be delivered at the leader and included in the *Local* message. Then by the Theorem 117, all the other remote clusters will receive a *Local* message for each cluster, including the current one. Thus, o will be delivered in the *Local* message and inserted to *operations*. Finally, after all the *Local* messages are delivered, o will be executed in the execution phase.

Case 2: o is from a remote cluster. Then we prove that o will be delivered at all the other clusters.

If o is from a remote cluster, then *operations* is only updated if Σ is valid and the deliver is for the same round. Σ is valid if it contains a quorum of commit certificate for each operation in *ops*: a quorum of commit messages certify the delivery in the local order protocol. *operations* is updated when receiving a valid *Local* message. Then by the Theorem 118 and Theorem 117, o will be delivered at all the other clusters through the same *Local* message.

■

Theorem 124 (Total order). For every pair of operations o and o', if a correct process executes only o, or executes o before o', then every correct process executes o' only after o.

Proof. We prove the total order property in two steps.

First we prove that if a pair of processes p_1 and p_2 both execute o and o', then they execute o and o' in the same order. Without loss of generality, let us assume that o is executed in p_1 before o' By Theorem 123, all the correct processes deliver the same operations for each cluster. Then they combine the operations in the predefined order based on the cluster identifier. Within each cluster, *ops* have been ordered across all the correct processes by the total order property of the underlying TOB protocol. Thus the combined operations keeps a total order across all the operations from all the clusters for round r: o is executed before o' at all correct processes including p_2.

In the second step, we prove by contradiction. Assume that process p' executed operation o' before operation o. The process p executed only o, or executed o before o'. In the case that it has executed only o and not o', then, by the Theorem 123, it will eventually execute o' after o. Thus, we will reach a state where p and p' have a different order for the two operations o and o', which contradicts the first statement. ■

Chapter 6

Conclusions

This dissertation explores the heterogeneity and openness of Byzantine replicated systems. More specifically, in the first thrust, I explore the research problem of how to partition and replicate data and computations so that end-to-end trustworthiness policies are guaranteed. In the second thrust, I investigate heterogeneous quorum systems where each process can declare its own quorums. I proved impossibility results for heterogeneous quorum systems, establish new sufficient conditions and propose abstract and concrete reliable broadcast and consensus protocols. Further, I design reconfiguration protocols that helps to maintain critical properties that are required for consensus. Lastly, I designed reconfiguration protocols for clustered replication system to enable heterogeneous and dynamic clustered replication.

In chapter 2, we introduce Hampa, a theoretical framework and a system for trustworthy distributed systems. We define a lattice model of resiliency, a security-typed object-based language to capture end-to-end type polices for the three aspects of trustworthiness

including confidentiality, integrity and availability. Further, we present a partitioning transformation, operational semantics, an information flow type inference system, and quorum constraint solving to automatically construct partitioned and replicated systems that guarantee non-interference and resiliency properties even in the face of Byzantine failures. Our experiments show that inferred distributed systems can gracefully tolerate attacks that are as strong as the specified policies.

In chapter 3, we dive into the formal definitions and specifications of heterogeneous quorum systems. We present a general model of heterogeneous quorum systems where each process defines its own set of quorums, and captured their properties. Through indistinguishably arguments, we proved that no deterministic quorum-based protocol can implement the consensus and Byzantine reliable broadcast abstractions on a heterogeneous quorum system that provides only quorum intersection and availability, which was previously proved to be necessary and sufficient for existing dissemination quorum systems. We introduce the quorum subsumption property, and show that the three conditions together are sufficient to implement the two abstractions. In order to show sufficiency, we present both abstract and concrete Byzantine broadcast and consensus protocols for heterogeneous quorum systems, and prove their correctness when the underlying quorum system maintain the three properties.

In chapter 4, we further investigate reconfiguration protocols and conditions for heterogeneous quorum systems. We revisit the model of heterogeneous quorum systems and present their graph characterization. In order to make them open, we addresses their reconfiguration. We prove trade-offs for the properties that reconfigurations can preserve,

and presents reconfiguration protocols with provable guarantees. We hope that this work further motivates the incorporation of open membership and heterogeneous trust into quorum systems, and helps blockchains avoid high energy consumption, and centralization at nodes with high computational power or stake.

In chapter 5, we examine heterogeneous clustered replicated system and their reconfiguration protocols. We present a protocol that adapts to different cluster sizes, and allows processes to join and leave clusters efficiently. Further, we state and prove the safety and liveness properties of the protocol.

Bibliography

[1] The coq proof assistant. https://coq.inria.fr/.

[2] Elc: Spacex lessons learned. https://lwn.net/Articles/540368/.

[3] Horizon api. https://developers.stellar.org/api. Accessed: 2023-05-01.

[4] LinkedIn's Voldemort. http://www.project-voldemort.com/.

[5] Memcached. http://memcached.org/.

[6] Pvs specification and verification system. http://pvs.csl.sri.com/.

[7] Stellarbeat.io. https://stellarbeat.io. Accessed: 2023-05-01.

[8] Stellarcore. https://github.com/stellar/stellar-core. Accessed: 2023-05-01.

[9] Stellardashboard. https://dashboard.stellar.org/. Accessed: 2023-05-01.

[10] grammer-v4. https://github.com/antlr/grammars-v4, 2017.

[11] Netty project. https://netty.io/, 2021.

[12] Daniel Abadi. Consistency tradeoffs in modern distributed database system design. *Computer*, 45(2), 2012.

[13] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 59–74, New York, NY, USA, 2005. ACM.

[14] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[15] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. *arXiv preprint arXiv:1612.02916*, 2016.

[16] Ittai Abraham, Kartik Nayak, and Nibesh Shrestha. Optimal good-case latency for rotating leader synchronous bft. *Cryptology ePrint Archive*, 2021.

[17] Ittai Abraham and Gilad Stern. Information theoretic hotstuff. *arXiv preprint arXiv:2009.12828*, 2020.

[18] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002.

[19] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[20] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Consensus with byzantine failures and little system synchrony. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 147–155. IEEE, 2006.

[21] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.

[22] Eduardo Adilio Pelinson Alchieri, Alysson Bessani, Fabiola Greve, and Joni da Silva Fraga. Knowledge connectivity requirements for solving byzantine consensus with unknown participants. *IEEE Transactions on Dependable and Secure Computing*, 15(2):246–259, 2016.

[23] Eduardo AP Alchieri, Alysson Neves Bessani, Joni da Silva Fraga, and Fab´**i**ola Greve. Byzantine consensus with unknown participants. In *International Conference On Principles Of Distributed Systems*, pages 22–40. Springer, 2008.

[24] Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 986–996, 2019.

[25] Orestis Alpos, Christian Cachin, and Luca Zanolini. How to trust strangers: Composition of byzantine quorum systems. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 120–131. IEEE, 2021.

[26] Peter Alvaro, Tyson Condie, Neil Conway, Joseph M. Hellerstein, and Russell Sears. I do declare: Consensus in a logic language. *SIGOPS Oper. Syst. Rev.*, 43(4):25–30, January 2010.

[27] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination analysis and placement for distributed programs. *ACM Trans. Database Syst.*, 42(4):23:1–23:31, October 2017.

[28] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in bloom: A calm and collected approach. In *In Proceedings 5th Biennial Conference on Innovative Data Systems Research*, pages 249–260, 2011.

[29] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, Michael K Reiter, and Rebecca N Wright. Dynamic byzantine quorum systems. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 283–292. IEEE, 2000.

[30] Yair Amir, Claudiu Danilov, Danny Dolev, Jonathan Kirsch, John Lane, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Steward: Scaling byzantine fault-tolerant replication to wide area networks. *IEEE Transactions on Dependable and Secure Computing*, 7(1):80–93, 2008.

[31] Mohammad Javad Amiri, Divyakant Agrawal, and Amr El Abbadi. Sharper: Sharding permissioned blockchains over network clusters. In *Proceedings of the 2021 international conference on management of data*, pages 76–88, 2021.

[32] Ignacio Amores-Sesar, Christian Cachin, and Jovana Mićić. Security analysis of ripple consensus. *arXiv preprint arXiv:2011.14816*, 2020.

[33] Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of tendermint-core blockchains. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[34] Zachary R. Anderson, David Gay, and Mayur Naik. Lightweight annotations for controlling sharing in concurrent data structures. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 98–109, 2009.

[35] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

[36] Appendix. Appendix. `https://www.cs.ucr.edu/~lesani/companion/popl19/Appendix.pdf`, 2018.

[37] Joe Armstrong. The development of erlang. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 196–203, New York, NY, USA, 1997. ACM.

[38] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and Padmanabhan Pillai. Meld: A declarative approach to programming ensembles. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2794–2800, Oct 2007.

[39] Michael P. Ashley-Rollman, Peter Lee, Seth Copen Goldstein, Padmanabhan Pillai, and Jason D. Campbell. A language for large ensembles of independently executing nodes. In *Proceedings of the 25th International Conference on Logic Programming*, ICLP '09, pages 265–280, Berlin, Heidelberg, 2009. Springer-Verlag.

[40] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM (JACM)*, 41(1):122–152, 1994.

[41] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2), 1994.

[42] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knezevic, Vivien Quema, and Marko Vukolic. The next 700 bft protocols. *ACM Trans. Comput. Syst.*, 32(4):12:1–12:45, January 2015.

[43] John Augustine, Valerie King, Anisur Rahaman Molla, Gopal Pandurangan, and Jared Saia. Scalable and Secure Computation Among Strangers: Message-Competitive Byzantine Protocols. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing (DISC 2020)*, volume 179 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:19, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[44] John Augustine, Gopal Pandurangan, and Peter Robinson. Fast byzantine leader election in dynamic networks. In *International Symposium on Distributed Computing*, pages 276–291. Springer, 2015.

[45] Michael Backes, Boris Köpf, and Andrey Rybalchenko. Automatic discovery and quantification of information leaks. In *2009 30th IEEE Symposium on Security and Privacy*, S&P '09, pages 141–153. IEEE, 2009.

[46] Michael Backes, Matteo Maffei, and Kim Pecina. Automated synthesis of privacy-preserving distributed applications. In *Proc. of ISOC NDSS*, NDSS '12, 2012.

[47] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014.

[48] Peter Bailis, Alan Fekete, Michael J Franklin, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1327–1342. ACM, 2015.

[49] Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. The potential dangers of causal consistency and an explicit solution. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 22. ACM, 2012.

[50] Peter Bailis and Ali Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *Communications of the ACM*, 56(5):55–63, 2013.

[51] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Bolt-on causal consistency. In *Proc. SIGMOD*, 2013.

[52] Peter Bailis, Shivaram Venkataraman, Michael J Franklin, Joseph M Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proceedings of the VLDB Endowment*, 5(8):776–787, 2012.

[53] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguica, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.

[54] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguica, Mahsa Najafzadeh, and Marc Shapiro. Towards fast invariant preservation in geo-replicated systems. *SIGOPS Oper. Syst. Rev.*, 49(1):121–125, January 2015.

[55] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Sok: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 183–198, 2019.

[56] Kshitij Bansal, Andrew Reynolds, Clark Barrett, and Cesare Tinelli. A new decision procedure for finite sets and cardinality constraints in smt. In *International Joint Conference on Automated Reasoning*, pages 82–98. Springer, 2016.

[57] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[58] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[59] Gilles Barthe, Pedro R D'argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[60] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, Francois Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 7, 2019.

[61] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. PRACTI replication. In *Proc. NSDI*, 2006.

[62] John Bender, Mohsen Lesani, and Jens Palsberg. Declarative fence insertion. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented*

Programming, Systems, Languages, and Applications, OOPSLA 2015, pages 367–385, New York, NY, USA, 2015. ACM.

[63] Giovanni Bernardi and Alexey Gotsman. Robustness against consistency models with atomic visibility. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 59. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[64] A. Bessani and P. Sousa. Smart — high-performance byzantine-fault-tolerant state machine replication. `http://code.google.com/p/bft-smart/`, 2009.

[65] Alysson Bessani, Marcel Santos, João Felix, Nuno Neves, and Miguel Correia. On the efficiency of durable state machine replication. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 169–180, Berkeley, CA, USA, 2013. USENIX Association.

[66] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. State machine replication for the masses with bft-smart. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, page 355–362, USA, 2014. IEEE Computer Society.

[67] Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with bft-smart. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

[68] Alysson Neves Bessani, Eduardo Pelison Alchieri, Miguel Correia, and Joni Silva Fraga. Depspace: A byzantine fault-tolerant coordination service. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 163–176, New York, NY, USA, 2008. ACM.

[69] João Paulo Bezerra, Petr Kuznetsov, and Alice Koroleva. Relaxed reliable broadcast for decentralized trust. *arXiv preprint arXiv:2109.08611*, 2021.

[70] João Paulo Bezerra, Petr Kuznetsov, and Alice Koroleva. Relaxed reliable broadcast for decentralized trust. In *Networked Systems: 10th International Conference, NETYS 2022, Virtual Event, May 17–19, 2022, Proceedings*, pages 104–118. Springer, 2022.

[71] Adithya Bhat, Nibesh Shrestha, Aniket Kate, and Kartik Nayak. Optrand: Optimistically responsive reconfigurable distributed randomness. In *NDSS*, 2023.

[72] Adithya Bhat, Nibesh Shrestha, Zhongtang Luo, Aniket Kate, and Kartik Nayak. Randpiper–reconfiguration-friendly random beacons with quadratic communication. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3502–3524, 2021.

[73] Mark Bickford. Component specification using event classes. In *Component-Based Software Engineering*, volume 5582 of *Lecture Notes in Computer Science*. 2009.

[74] M. Biely, P. Delgado, Z. Milosevic, and A. Schiper. Distal: A framework for implementing fault-tolerant distributed algorithms. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–8, June 2013.

[75] Kenneth P. Birman. Replication and fault-tolerance in the ISIS system. In *Proc. SOSP*, 1985.

[76] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst.*, 5(1), 1987.

[77] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. In *Proc. POPL*, 2006.

[78] Ranadeep Biswas, Michael Emmi, and Constantin Enea. On the complexity of checking consistency for replicated data types. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*, pages 324–343. Springer, 2019.

[79] S. Biswas, J. Huang, Sengupta A., , and Bond M. D. Doublechecker: Efficient sound and precise atomicity checking. In *Proc. PLDI*, 2014.

[80] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA*, 2006.

[81] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proc. ICSE*, 2011.

[82] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 141–154, Berkeley, CA, USA, 2011. USENIX Association.

[83] Malte Borcherding. Levels of authentication in distributed agreement. In *International Workshop on Distributed Algorithms*, pages 40–55. Springer, 1996.

[84] A. Bouajjani, C. Enea, and J. Hamza. Verifying eventual consistency of optimistic replication systems. In *Proc. POPL*, 2014.

[85] Andrea Bracciali, Davide Grossi, and Ronald de Haan. Decentralization in open quorum systems: Limitative results for ripple and stellar. In *2nd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[86] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[87] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Liveness and latency of byzantine state-machine replication. In *36th International Symposium on Distributed Computing (DISC 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[88] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. *Distributed Computing*, 35(6):503–532, 2022.

[89] Manuel Bravo and Alexey Gotsman. Reconfigurable atomic transaction commit. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 399–408, 2019.

[90] Coen Bron and Joep Kerbosch. Algorithm 457: Finding all cliques of an undirected graph. *Commun. ACM*, 16(9):575–577, September 1973.

[91] Stephen Brookes and Peter W. O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, August 2016.

[92] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, volume 57, 2004.

[93] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. Serializability for eventual consistency: Criterion, analysis, and applications. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 458–472, New York, NY, USA, 2017. ACM.

[94] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains.* PhD thesis, University of Guelph, 2016.

[95] Ethan Buchman, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, Dragos-Adrian Seredinschi, and Josef Widder. Revisiting tendermint: Design tradeoffs, accountability, and practical use. In *2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 11–14. IEEE, 2022.

[96] Sebastian Burckhardt. *Principles of Eventual Consistency.* Foundations and Trends in Programming Languages. 2014.

[97] Sebastian Burckhardt, Alexey Gotsman, Hongseok Yang, and Marek Zawirski. Replicated data types: Specification, verification, optimality. In *Proc. POPL*, 2014.

[98] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proc. ASE*, 2008.

[99] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[100] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

[101] Christian Cachin, Rachid Guerraoui, and Lu´*i*s Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

[102] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[103] Christian Cachin, Giuliano Losa, and Luca Zanolini. Quorum systems in permissionless network. *arXiv preprint arXiv:2211.05630*, 2022.

[104] Christian Cachin and Björn Tackmann. Asymmetric distributed trust. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[105] Christian Cachin and Luca Zanolini. From symmetric to asymmetric asynchronous byzantine consensus. *arXiv preprint arXiv:2005.08795*, 2020.

[106] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: Automatically generating inputs of death. In *Proc. CCS*, 2006.

[107] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam Mckelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Marvin Mcnett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *In SOSP '11*, pages 143–157, 2011.

[108] Domenico Cantone, Eugenio Omodeo, and Alberto Policriti. *Set theory for computing: from decision procedures to declarative programming with sets.* Springer Science & Business Media, 2013.

[109] Domenico Cantone and Calogero G Zarba. A new fast tableau-based decision procedure for an unquantified fragment of set theory. In *Automated Deduction in Classical and Non-Classical Logics*, pages 126–136. Springer, 2000.

[110] Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. Towards formal verification of hotstuff-based byzantine fault tolerant consensus in agda. In *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*, pages 616–635. Springer, 2022.

[111] Nuno Carvalho and et. al. Appia framework. `http://appia.di.fc.ul.pt/wiki/index.php?title=Main_Page`, 2011. Accessed: 2018-06-23.

[112] Darion Cassel, Yan Huang, and Limin Jia. Flownotation: An annotation system for statically enforcing information flow policies in c. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 2207–2209, 2018.

[113] M. Castro and B. Liskov. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Cambridge, MA, USA, 1999.

[114] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 173–186, Berkeley, CA, USA, 1999. USENIX Association.

[115] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, November 2002.

[116] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[117] David Cavin, Yoav Sasson, and André Schiper. Consensus with unknown participants or fundamental self-organization. In *International Conference on Ad-Hoc Networks and Wireless*, pages 135–148. Springer, 2004.

[118] Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 42. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[119] Saksham Chand, Yanhong A Liu, and Scott D Stoller. Formal verification of multi-paxos for distributed consensus. In *International Symposium on Formal Methods*, pages 119–136. Springer, 2016.

[120] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[121] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 322–330, 1996.

[122] Feng Chen and Grigore Rosu. Parametric and sliced causality. In *Proc. CAV*, 2007.

[123] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. jpredictor: A predictive runtime analysis tool for java. In *Proc. ICSE*, 2008.

[124] Qichang Chen, Liqiang Wang, Zijiang Yang, and ScottD. Stoller. Have: Detecting atomicity violations via integrated dynamic and static analysis. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering*, volume 5503 of *LNCS*, pages 425–439. Springer Berlin Heidelberg, 2009.

[125] Alvin Cheung, Owen Arden, Samuel Madden, and Andrew C Myers. Automatic partitioning of database applications. *Proceedings of the VLDB Endowment*, 5(11), 2012.

[126] Alvin Cheung, Owen Arden, Samuel Madden, Armando Solar-Lezama, and Andrew C Myers. Statusquo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[127] Adam Chlipala. Mostly-automated verification of low-level programs in computational separation logic. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 234–245, New York, NY, USA, 2011. ACM.

[128] Gregory V Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

[129] Stephen Chong, Jed Liu, Andrew C Myers, Xin Qi, Krishnaprasad Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. *ACM SIGOPS Operating Systems Review*, 41(6):31–44, 2007.

[130] Kevin Clancy and Heather Miller. Monotonicity types for distributed dataflow. In *Proceedings of the Programming Models and Languages for Distributed Computing*, page 2. ACM, 2017.

[131] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 277–290, New York, NY, USA, 2009. ACM.

[132] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, 2009.

[133] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[134] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[135] Shir Cohen, Idit Keidar, and Oded Naor. Byzantine agreement with less communication: Recent advances. *ACM SIGACT News*, 52(1):71–80, 2021.

[136] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[137] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2), 2008.

[138] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[139] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[140] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[141] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190, 2006.

[142] Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In *Advances in Cryptology–ASIACRYPT 2007: 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007. Proceedings 13*, pages 357–375. Springer, 2007.

[143] Hung Dang, Tien Tuan Anh Dinh, Dumitrel Loghin, Ee-Chien Chang, Qian Lin, and Beng Chin Ooi. Towards scaling blockchain systems via sharding. In *Proceedings of the 2019 international conference on management of data*, pages 123–140, 2019.

[144] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[145] 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 *Proc. SOSP*, 2007.

[146] Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 338–346, 2004.

[147] Dorothy E Denning and Peter J Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, 1977.

[148] Stellar Developer. Stellar configuring. https://developers.stellar.org/docs/run-core-node/configuring, 2022.

[149] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *2010 IEEE Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.

[150] Colin Dixon, Hardeep Uppal, Vjekoslav Brajkovic, Dane Brandon, Thomas Anderson, and Arvind Krishnamurthy. Ettm: A scalable fault tolerant network manager. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 85–98, Berkeley, CA, USA, 2011. USENIX Association.

[151] Cezara Dragoi, Thomas A Henzinger, and Damien Zufferey. PSYNC : A partially synchronous language for fault-tolerant distributed algorithms. *Popl*, pages 1–16, 2016.

[152] Cezara Drăgoi, Thomas A Henzinger, and Damien Zufferey. Psync: a partially synchronous language for fault-tolerant distributed algorithms. *ACM SIGPLAN Notices*, 51(1):400–415, 2016.

[153] K. Driscoll, B. Hall, M. Paulitsch, P. Zumsteg, and H. Sivencrona. The real byzantine generals. In *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, volume 2, pages 6.D.4–61–11 Vol.2, Oct 2004.

[154] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. *Byzantine Fault Tolerance, from Theory to Reality*, pages 235–248. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[155] Kevin R Driscoll. *Murphy Was an Optimist*, pages 481–482. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[156] Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. Bchain: Byzantine replication with high throughput and embedded reconfiguration. In *Principles of Distributed Systems: 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings 18*, pages 91–106. Springer, 2014.

[157] Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041, 2018.

[158] Sisi Duan and Haibin Zhang. Foundations of dynamic bft. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1546–1546. IEEE Computer Society, 2022.

[159] Bruno Dutertre. Yices 2.2. In *Proc. CAV*, 2014.

[160] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[161] S. Elnikety, F. Pedone, and W. Zwaenepoel. Database replication using generalized snapshot isolation. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 73–84, Oct 2005.

[162] Michael Emmi and Constantin Enea. Monitoring weak consistency. In *Proc. CAV*, 2018.

[163] Mahdi Eslamimehr and Jens Palsberg. Race directed scheduling of concurrent programs. In *Proc. PPoPP*, 2014.

[164] Azadeh Farzan and P. Madhusudan. Causal atomicity. In Thomas Ball and RobertB. Jones, editors, *Computer Aided Verification*, volume 4144 of *LNCS*, pages 315–328. Springer Berlin Heidelberg, 2006.

[165] Azadeh Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In *Proc. CAV*, 2008.

[166] Azadeh Farzan and P. Madhusudan. The complexity of predicting atomicity violations. In *Proc. TACAS*, 2009.

[167] Azadeh Farzan, P. Madhusudan, and Francesco Sorrentino. Meta-analysis for atomicity violations under nested locking. In *Proc. CAV*, 2009.

[168] Alan Fekete. Allocating isolation levels to transactions. In *Proceedings of the Twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '05, pages 206–215, New York, NY, USA, 2005. ACM.

[169] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, June 2005.

[170] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1), 1988.

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

[172] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[173] 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.

[174] Cormac Flanagan. Verifying commit-atomicity using model-checking. In Susanne Graf and Laurent Mounier, editors, *Model Checking Software*, volume 2989 of *LNCS*, pages 252–266. Springer Berlin Heidelberg, 2004.

[175] Cormac Flanagan and Stephen N Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proc. POPL*, 2004.

[176] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, August 2008.

[177] Cormac Flanagan, Stephen N. Freund, and Jaeheon Yi. Velodrome: A sound and complete dynamic atomicity checker for multithreaded programs. In *Proc. PLDI*, 2008.

[178] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proc. PLDI*, 2003.

[179] Mitch Fletcher. Progression of an open architecture: from orion to altair and lss. Technical Report White paper S65- 5000-20-0, Honeywell International, Glendale, 2009.

[180] Martin Florian, Sebastian Henningsen, Charmaine Ndolo, and Björn Scheuermann. The sum of its parts: Analysis of federated byzantine agreement systems. *Distributed Computing*, pages 1–19, 2022.

[181] Cédric Fournet, Gurvan Le Guernic, and Tamara Rezk. A security-preserving compiler for distributed programs: From information-flow policies to cryptographic mechanisms. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 432–441, 2009.

[182] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1187–1201, 2022.

[183] Juan Garay and Aggelos Kiayias. Sok: A consensus taxonomy in the blockchain era. In *Cryptographers' track at the RSA conference*, pages 284–318. Springer, 2020.

[184] Rui Garcia, Rodrigo Rodrigues, and Nuno Preguica. Efficient middleware for byzantine fault tolerant database replication. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 107–122, New York, NY, USA, 2011. ACM.

[185] Álvaro Garc´**i**a-Pérez and Alexey Gotsman. Federated byzantine quorum systems. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[186] Álvaro Garc´**i**a-Pérez, Alexey Gotsman, Yuri Meshman, and Ilya Sergey. Paxos consensus, deconstructed and abstracted. In *European Symposium on Programming*, pages 912–939. Springer, Cham, 2018.

[187] Álvaro Garc´**i**a-Pérez and Maria A Schett. Deconstructing stellar consensus (extended version). *arXiv preprint arXiv:1911.05145*, 2019.

[188] Adria Gascón and Ashish Tiwari. Synthesis of a simple self-stabilizing system. In *Proc. 3rd Workshop on Synthesis (SYNT)*, 2014.

[189] André Gaul, Ismail Khoffi, Jörg Liesen, and Torsten Stüber. Mathematical analysis and algorithms for federated byzantine agreement systems. *arXiv preprint arXiv:1912.01365*, 2019.

[190] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 21–21, Berkeley, CA, USA, 2007. USENIX Association.

[191] Rati Gelashvili, Lefteris Kokoris-Kogias, Alexander Spiegelman, and Zhuolun Xiang. Brief announcement: Be prepared when network goes bad: An asynchronous view-change protocol. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 187–190, 2021.

[192] Arthur Gervais, Ghassan O Karame, Karl Wüst, Vasileios Glykantzis, Hubert Ritzdorf, and Srdjan Capkun. On the security and performance of proof of work blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 3–16, 2016.

[193] David K Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM, 1979.

[194] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th symposium on operating systems principles*, pages 51–68, 2017.

[195] Seth Gilbert. *RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks.* PhD thesis, Massachusetts Institute of Technology, 2003.

[196] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), June 2002.

[197] Seth Gilbert and Nancy A. Lynch. Perspectives on the CAP theorem. *IEEE Computer*, 45(2):30–36, 2012.

[198] Guy Goren, Yoram Moses, and Alexander Spiegelman. Probabilistic indistinguishability and the quality of validity in byzantine agreement. *arXiv preprint arXiv:2011.04719*, 2020.

[199] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. ACM.

[200] Mike Graf, Ralf Küsters, and Daniel Rausch. Accountability in a permissioned blockchain: Formal analysis of hyperledger fabric. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 236–255. IEEE, 2020.

[201] Mike Graf, Daniel Rausch, Viktoria Ronge, Christoph Egger, Ralf Küsters, and Dominique Schröder. A security framework for distributed ledgers. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1043–1064, 2021.

[202] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang, and Yu Guo. Deep specifications and certified abstraction layers. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 595–608, New York, NY, USA, 2015. ACM.

[203] Marco Guarnieri, Musard Balliu, Daniel Schoepe, David Basin, and Andrei Sabelfeld. Information-flow control for database-backed applications. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 79–94. IEEE, 2019.

[204] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376, 2010.

[205] Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic byzantine reliable broadcast [technical report]. *arXiv preprint arXiv:2001.06271*, 2020.

[206] Rachid Guerraoui, Matej Pavlovic, and Dragos-Adrian Seredinschi. Incremental consistency guarantees for replicated objects. In *OSDI*, pages 169–184, 2016.

[207] Guy Golan Gueta, Ittai Abraham, Shelly Grossman, Dahlia Malkhi, Benny Pinkas, Michael Reiter, Dragos-Adrian Seredinschi, Orr Tamir, and Alin Tomescu. Sbft: A scalable and decentralized trust infrastructure. In *2019 49th Annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 568–580. IEEE, 2019.

[208] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous bft closer to practice. *Cryptology ePrint Archive*, 2022.

[209] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818, 2020.

[210] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: replication at the speed of multi-core. In *Proc. Eurosys*, 2014.

[211] Zhenyu Guo, Sean McDirmid, Mao Yang, Li Zhuang, Pu Zhang, Yingwei Luo, Tom Bergan, Peter Bodik, Madan Musuvathi, Zheng Zhang, and Lidong Zhou. Failure recovery: When the cure is worse than the disease. In *Proc. HotOS*, 2013.

[212] Suyash Gupta, Sajjad Rahnama, Jelle Hellings, and Mohammad Sadoghi. Resilientdb: Global scale resilient blockchain fabric. *Proceedings of the VLDB Endowment*, 13(6).

[213] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, February 2009.

[214] Md. E. Haque, Yong Hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn S. McKinley. Few-to-many: Incremental parallelism for reducing tail latency in interactive services. In *Proc. ASPLOS*, 2015.

[215] John Hatcliff, Robby, and MatthewB. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 175–190. Springer Berlin Heidelberg, 2004.

[216] Klaus Havelund and Thomas Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[217] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: Proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[218] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.

[219] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 165–181, Berkeley, CA, USA, 2014. USENIX Association.

[220] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In *Proc. ESOP*, 1986.

[221] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proc. ISCA*, 1993.

[222] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[223] Martin Hirt, Ard Kastrati, and Chen-Da Liu-Zhang. Multi-threshold asynchronous reliable broadcast and consensus. *Cryptology ePrint Archive*, 2020.

[224] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[225] Jan Hoffmann, Michael Marmar, and Zhong Shao. Quantitative reasoning for proving lock-freedom. In *Proceedings of the 2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 124–133. IEEE Computer Society, 2013.

[226] Brandon Holt, James Bornholt, Irene Zhang, Dan Ports, Mark Oskin, and Luis Ceze. Disciplined inconsistency with consistency types. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 279–293, New York, NY, USA, 2016. ACM.

[227] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[228] Phillip W. Hutto and Mustaque Ahamad. Slow memory: Weakening consistency to enchance concurrency in distributed shared memories. In *ICDCS*, 1990.

[229] Rebecca Ingram, Patrick Shields, Jennifer E Walter, and Jennifer L Welch. An asynchronous leader election algorithm for dynamic networks. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

[230] Sushil Jajodia and Catherine A Meadows. Mutual consistency in decentralized distributed systems. In *1987 IEEE Third International Conference on Data Engineering*, pages 396–404. IEEE, 1987.

[231] Mauro Jaskelioff and Stephan Merz. Proving the correctness of disk paxos. *Archive of Formal Proofs*, 2005, 2005.

[232] Chamikara Jayalath and Patrick Eugster. Efficient geo-distributed data processing with rout. In *Proc. ICDCS*, 2013.

[233] Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.

[234] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soule, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, 2018.

[235] Annu John, Igor Konnov, Ulrich Schmid, Helmut Veith, and Josef Widder. Parameterized model checking of fault-tolerant distributed algorithms by abstraction. In *Proc. FMCAD*, 2013.

[236] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[237] Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. Alone together: Compositional reasoning and inference for weak isolation. *Proc. ACM Program. Lang.*, 2(POPL):27:1–27:34, December 2017.

[238] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. Cheapbft: Resource-efficient byzantine fault tolerance. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 295–308, New York, NY, USA, 2012. ACM.

[239] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is dag. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 165–175, 2021.

[240] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III 40*, pages 451–480. Springer, 2020.

[241] Pertti Kellomäki. An annotated specification of the consensus protocol of paxos using superposition in pvs. Technical report, Technical report 36, Tampere University of Technology, Institute of Software . . . , 2004.

[242] Christoph Kerschbaumer, Eric Hennigan, Per Larsen, Stefan Brunthaler, and Michael Franz. Crowdflow: Efficient information flow security. In *Information Security*, pages 321–337. Springer, 2015.

[243] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[244] Minjeong Kim, Yujin Kwon, and Yongdae Kim. Is stellar as secure as you think? In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 377–385. IEEE, 2019.

[245] J. Kirsch, S. Goose, Y. Amir, D. Wei, and P. Skare. Survivable scada via intrusion-tolerant replication. *IEEE Transactions on Smart Grid*, 5(1):60–70, Jan 2014.

[246] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, February 2014.

[247] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

[248] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 351–364, Berkeley, CA, USA, 2010. USENIX Association.

[249] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58, New York, NY, USA, 2007. ACM.

[250] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

[251] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009.

[252] Philipp Kufner, Uwe Nestmann, and Christina Rickmann. Formal verification of distributed algorithms. In *Theoretical Computer Science*, volume 7604 of *LNCS*. 2012.

[253] Viktor Kuncak, Huu Hai Nguyen, and Martin Rinard. Deciding boolean algebra with presburger arithmetic. *Journal of Automated Reasoning*, 36(3):213–239, 2006.

[254] Viktor Kuncak and Martin Rinard. Towards efficient satisfiability checking for boolean algebra with presburger arithmetic. In *International Conference on Automated Deduction*, pages 215–230. Springer, 2007.

[255] Łukasz Lachowski. Complexity of the quorum intersection property of the federated byzantine agreement system. *arXiv preprint arXiv:1902.06493*, 2019.

[256] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4), 1992.

[257] Ori Lahav and Viktor Vafeiadis. Owicki-gries reasoning for weak memory models. In *International Colloquium on Automata, Languages, and Programming*, pages 311–323. Springer, 2015.

[258] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), 2010.

[259] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[260] Leslie Lamport. Introduction to TLA. Technical report, 1994.

[261] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.

[262] LESLIE LAMPORT. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[263] Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

[264] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[265] Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19:104–125, 2006.

[266] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, 2010.

[267] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, 1982.

[268] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[269] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[270] Mohsen Lesani. Tm testing tool source code. `http://people.csail.mit.edu/lesani/companion/disc13/index.html`, 2013.

[271] Mohsen Lesani. Pvs proofs of tl2 transactional memory algorithm based on sol logic. `http://people.csail.mit.edu/lesani/companion/dissertation/`, 2014.

[272] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 357–370, New York, NY, USA, 2016. ACM.

[273] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *Proceedings of the 23rd International Conference on Concurrency Theory*, CONCUR'12, pages 516–530, Berlin, Heidelberg, 2012. Springer-Verlag.

[274] Mohsen Lesani, Victor Luchangco, and Mark Moir. Pvs framework for transactional memory verification. `http://people.csail.mit.edu/lesani/companion/concur12/index.html`, 2014.

[275] Mohsen Lesani, Todd Millstein, and Jens Palsberg. Chapar verification framework source code. `http://people.csail.mit.edu/lesani/companion/popl16/artifact/index.html`, 2014.

[276] Mohsen Lesani, Todd Millstein, and Jens Palsberg. Snowflake verification tool source code. `http://people.csail.mit.edu/lesani/companion/cav14/`, 2014.

[277] Mohsen Lesani, Todd D. Millstein, and Jens Palsberg. Automatic atomicity verification for clients of concurrent data structures. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 550–567, 2014.

[278] Mohsen Lesani and Jens Palsberg. Communicating memory transactions. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 157–168, New York, NY, USA, 2011. ACM.

[279] Mohsen Lesani and Jens Palsberg. *Proving Non-opacity*, pages 106–120. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[280] Mohsen Lesani and Jens Palsberg. *Decomposing Opacity*, pages 391–405. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.

[281] Andrew Lewis-Pye and Tim Roughgarden. Byzantine generals in the permissionless setting. *arXiv preprint arXiv:2101.07095*, 2021.

[282] Andrew Lewis-Pye and Tim Roughgarden. Permissionless consensus. *arXiv preprint arXiv:2304.14701*, 2023.

[283] Cheng Li, João Leitão, Allen Clement, Nuno Preguica, Rodrigo Rodrigues, and Viktor Vafeiadis. Automating the choice of consistency levels in replicated systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 281–292, Berkeley, CA, USA, 2014. USENIX Association.

[284] Cheng Li, João Leitão, Allen Clement, Nuno Preguica, and Rodrigo Rodrigues. Minimizing coordination in replicated systems. In *Proceedings of the First Workshop on Principles and Practice of Consistency for Distributed Data*, page 8. ACM, 2015.

[285] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguica, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[286] Jinyuan Li and David Maziéres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 10–10, Berkeley, CA, USA, 2007. USENIX Association.

[287] Jinyuan Li and David Mazieres. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI*, pages 10–10, 2007.

[288] Peixuan Li and Danfeng Zhang. A derivation framework for dependent security label inference. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–26, 2018.

[289] Xiao Li, Eric Chan, and Mohsen Lesani. Quorum subsumption for heterogeneous quorum systems. In *37th International Symposium on Distributed Computing (DISC 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

[290] Xiao Li, Eric Chan, and Mohsen Lesani. Quorum subsumption for heterogeneous quorum systems. technical report. In *International Symposium on Distributed Computing (DISC 2023)*, 2023.

[291] Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hampa: Solver-aided recency-aware replication. In *International Conference on Computer Aided Verification*, pages 324–349. Springer, 2020.

[292] Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hamraz: Resilient partitioning and replication. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2267–2284. IEEE, 2022.

[293] Xiao Li and Mohsen Lesani. Reconfigurable heterogeneous quorum systems. `https://mohsenlesani.github.io/companion/disc24/FullPaper.pdf`.

[294] Xiao Li and Mohsen Lesani. Open heterogeneous quorum systems, 2023.

[295] Xiao Li and Mohsen Lesani. Reconfigurable heterogeneous quorum systems. *arXiv preprint: arXiv: 2304.02156*, 2024.

[296] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

[297] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the harp file system. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 226–238, New York, NY, USA, 1991. ACM.

[298] Mark C Little and Daniel L McCue. The replica management system: a scheme for flexible and dynamic replication. In *Proceedings of 2nd International Workshop on Configurable Distributed Systems*, pages 46–57. IEEE, 1994.

[299] Jed Liu, Owen Arden, Michael D George, and Andrew C Myers. Fabric: Building open distributed systems securely by construction. *Journal of Computer Security*, 25(4-5):367–426, 2017.

[300] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 503–517, Berkeley, CA, USA, 2014. USENIX Association.

[301] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2359–2371, 2017.

[302] 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 *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 423–437, Berkeley, CA, USA, 2008. USENIX Association.

[303] Yanhong A. Liu, Scott D. Stoller, Bo Lin, and Michael Gorbovitski. From clarity to efficiency for distributed algorithms. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 395–410, New York, NY, USA, 2012. ACM.

[304] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. SOSP*, 2011.

[305] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. NSDI*, 2013.

[306] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't settle for eventual consistency. *Communications of the ACM*, 57(5):61–68, 2014.

[307] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 80–96, 2019.

[308] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. STOC '14, pages 1219–1234, 2012.

[309] Jacob R Lorch, Atul Adya, William J Bolosky, Ronnie Chaiken, John R Douceur, and Jon Howell. The smart way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 103–115, 2006.

[310] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The smart way to migrate replicated stateful services. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 103–115, New York, NY, USA, 2006. ACM.

[311] Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[312] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 887–903, 2019.

[313] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proc. ASPLOS*, 2008.

[314] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. Avio: Detecting atomicity violations via access interleaving invariants. In *Proc. ASPLOS*, 2006.

[315] Shiyong Lu, Arthur Bernstein, and Philip Lewis. Correct execution of transactions at different isolation levels. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1070–1081, 2004.

[316] Loi Luu, Viswesh Narayanan, Chaodong Zheng, Kunal Baweja, Seth Gilbert, and Prateek Saxena. A secure sharding protocol for open blockchains. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 17–30, 2016.

[317] Nancy Lynch and Alex A Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Distributed Computing: 16th International Conference, DISC 2002 Toulouse, France, October 28–30, 2002 Proceedings 16*, pages 173–190. Springer, 2002.

[318] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2, 1989.

[319] Ethan MacBrough. Cobalt: Bft governance in open networks. *arXiv preprint arXiv:1802.07240*, 2018.

[320] P. Madhusudan and P.S. Thiagarajan. Distributed controller synthesis for local specifications. In Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen, editors, *Automata, Languages and Programming*, pages 396–407, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[321] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical Report UTCS TR-11-22, The University of Texas at Austin, 2011.

[322] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and Parthasarathy Madhusudan. Verifying security invariants in expressos. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 293–304, New York, NY, USA, 2013. ACM.

[323] Ivan Malakhov, Andrea Marin, Sabina Rossi, and Daria Smuseva. On the use of proof-of-work in permissioned blockchains: Security and fairness. *IEEE Access*, 10:1305–1316, 2021.

[324] Dahlia Malkhi and Kartik Nayak. Hotstuff-2: Optimal two-phase responsive bft. *Cryptology ePrint Archive*, 2023.

[325] Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1041–1053, 2019.

[326] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[327] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[328] Mohammad Hossein Manshaei, Murtuza Jadliwala, Anindya Maiti, and Mahdi Fooladgar. A game-theoretic analysis of shard-based permissionless blockchains. *IEEE Access*, 6:78100–78112, 2018.

[329] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for wans. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 369–384, Berkeley, CA, USA, 2008. USENIX Association.

[330] Parisa Jalili Marandi, Samuel Benz, Fernando Pedone, and Kenneth P. Birman. The performance of Paxos in the cloud. In *Proc. SRDS*, 2014.

[331] Parisa Jalili Marandi, M. Primi, N. Schiper, and F. Pedone. Ring paxos: A high-throughput atomic broadcast protocol. In *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*, pages 527–536, June 2010.

[332] Giorgia Azzurra Marson, Sebastien Andreina, Lorenzo Alluminio, Konstantin Munichev, and Ghassan Karame. Mitosis: practically scaling permissioned blockchains. In *Annual Computer Security Applications Conference*, pages 773–783, 2021.

[333] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, 1989.

[334] David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32:1–45, 2015.

[335] Baoluo Meng, Andrew Reynolds, Cesare Tinelli, and Clark Barrett. Relational constraint solving in smt. In *International Conference on Automated Deduction*, pages 148–165. Springer, 2017.

[336] Matthew Milano and Andrew C Myers. Mixt: A language for mixing consistency in geodistributed transactions. 2018.

[337] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 31–42, New York, NY, USA, 2016. Association for Computing Machinery.

[338] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

[339] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, September 1992.

[340] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100(1):41–77, September 1992.

[341] Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1686–1699, 2021.

[342] Atsuki Momose and Ling Ren. Constant latency in sleepy consensus. *Cryptology ePrint Archive*, 2022.

[343] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 358–372, New York, NY, USA, 2013. ACM.

[344] Achour Mostefaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and Amr EL Abbadi. From static distributed systems to dynamic systems. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 109–118. IEEE, 2005.

[345] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In *International Conference on Certified Programs and Proofs*, pages 126–142. Springer, 2012.

[346] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proc. NSDI*, 2004.

[347] Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

[348] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The cise tool: Proving weakly-consistent applications correct. In *Proceedings of the 2Nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, pages 2:1–2:3, New York, NY, USA, 2016. ACM.

[349] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System.

[350] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *White paper*, 2008.

[351] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *arXiv preprint arXiv:1909.05204*, 2019.

[352] Sri Hari Krishna Narayanan, Mahmut Kandemir, and R Brooks. Performance aware secure code partitioning. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pages 1–6. IEEE, 2007.

[353] Francesco Zappa Nardelli, Peter Sewell, Jaroslav Sevcik, Susmit Sarkar, Scott Owens, Luc Maranget, Mark Batty, and Jade Alglave. Relaxed memory models must be rigorous. In *Exploiting Concurrency Efficiently and Correctly Workshop*, 2009.

[354] Alejandro Naser-Pastoriza, Gregory Chockler, and Alexey Gotsman. Fault-tolerant computing with unreliable channels. In *27th International Conference on Principles of Distributed Systems (OPODIS 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2024.

[355] Joachim Neu, Ertem Nusret Tas, and David Tse. Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 446–465. IEEE, 2021.

[356] Matthias Neubauer and Peter Thiemann. From sequential programs to multi-tier applications by program transformation. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 221–232, 2005.

[357] Peter W. OHearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

[358] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[359] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[360] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[361] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, 1976.

[362] Oded Padon, Giuliano Losa, Mooly Sagiv, and Sharon Shoham. Paxos made epr: decidable reasoning about distributed protocols. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–31, 2017.

[363] Chang-Seo Park and Koushik Sen. Randomized active atomicity violation detection in concurrent programs. In *Proc. FSE*, 2008.

[364] James Parker, Niki Vazou, and Michael Hicks. Lweb: Information flow security for multi-tier web applications. volume 3 of *POPL '19*, pages 1–30. ACM New York, NY, USA, 2019.

[365] Terence Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[366] Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. *Cryptology ePrint Archive*, 2016.

[367] Rafael Pass and Elaine Shi. Thunderella: Blockchains with optimistic instant confirmation. In *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37*, pages 3–33. Springer, 2018.

[368] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 207–218, New York, NY, USA, 2013. ACM.

[369] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proc. SOSP*, 1997.

[370] James L. Peterson. Petri nets. *ACM Comput. Surv.*, 9(3):223–252, September 1977.

[371] F. Pfenning and C. Elliot. Higher-order abstract syntax. In *Proc. PLDI*, 1988.

[372] Frank Pfenning and Robert J. Simmons. Substructural operational semantics as ordered logic programming. In *Proc. LICS*, 2009.

[373] Gordon D. Plotkin. The origins of structural operational semantics. In *Proc. Journal of Logic and Algebraic Programming*, pages 60–61, 2004.

[374] Francois Pottier and Vincent Simonet. Information flow inference for ml. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, 2002.

[375] Vincent Rahli. Interfacing with proof assistants for domain specific programming using EventML. 10th International Workshop on User Interfaces for Theorem Provers, 2012.

[376] Vincent Rahli, David Guaspari, Mark Bickford, and Robert L Constable. Formal specification, verification, and implementation of fault-tolerant systems using eventml. *Electronic Communications of the EASST*, 72, 2015.

[377] Vincent Rahli, Ivana Vukotic, Marcus Völp, and Paulo Esteves-Verissimo. Velisarios: Byzantine fault-tolerant protocols powered by coq. In *European Symposium on Programming*, pages 619–650. Springer, 2018.

[378] Tayebeh Rajabi, Alvi Ataur Khalil, Mohammad Hossein Manshaei, Mohammad Ashiqur Rahman, Mohammad Dakhilalian, Maurice Ngouen, Murtuza Jadliwala, and A Selcuk Uluagac. Feasibility analysis for sybil attacks in shard-based permissionless blockchains. *Distributed Ledger Technologies: Research and Practice*, 2(4):1–21, 2023.

[379] Vineet Rajani and Deepak Garg. On the expressiveness and semantics of information flow types. *Journal of Computer Security*, (Preprint):1–28, 2020.

[380] Krithi Ramamritham and Calton Pu. A formal characterization of epsilon serializability. *IEEE Transactions on Knowledge and Data Engineering*, 7(6):997–1007, 1995.

[381] Aseem Rastogi, Matthew A Hammer, and Michael Hicks. Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy*, pages 655–670. IEEE, 2014.

[382] Michel Raynal and André Schiper. From causal consistency to sequential consistency in shared memory systems. volume 1026 of *LNCS*. 1995.

[383] Michael K Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, 1996.

[384] Daniel Ricketts, Valentin Robert, Dongseok Jang, Zachary Tatlock, and Sorin Lerner. Automating formal proofs for reactive systems. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 452–462, New York, NY, USA, 2014. ACM.

[385] Tom Ridge. Verifying distributed systems: the operational approach. In *Proc. POPL*, 2009.

[386] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel Distrib. Comput.*, 71(3), 2011.

[387] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1311–1326, New York, NY, USA, 2015. ACM.

[388] John Rushby. *Bus Architectures for Safety-Critical Embedded Systems*, pages 306–323. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[389] Muhammad Saad, Songqing Chen, and David Mohaisen. Syncattack: Double-spending in bitcoin without mining power. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1668–1685, 2021.

[390] Muhammad Saad and David Mohaisen. Three birds with one stone: Efficient partitioning attacks on interdependent cryptocurrency networks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1404–1418. IEEE Computer Society, 2022.

[391] Muhammad Saad and David Mohaisen. Three birds with one stone: Efficient partitioning attacks on interdependent cryptocurrency networks. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 111–125. IEEE, 2023.

[392] Andrei Sabelfeld and Heiko Mantel. Static confidentiality enforcement for distributed programs. In *International Static Analysis Symposium*, pages 376–394. Springer, 2002.

[393] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003.

[394] Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In *Proc. ESOP*, 2009.

[395] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997.

[396] N. Schiper, V. Rahli, R. van Renesse, M. Bickford, and R.L. Constable. Developing correctly replicated databases using formal tools. In *Proc. DSN*, 2014.

[397] David Schwartz, Noah Youngs, and Arthur Britto. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5(8):151, 2014.

[398] Alessandro Sforzin, Matteo Maso, Claudio Soriente, and Ghassan Karame. On the storage overhead of proof-of-work blockchains. In *2022 IEEE International Conference on Blockchain (Blockchain)*, pages 258–265. IEEE, 2022.

[399] Marc Shapiro, Nuno Preguica, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report RR-7506, INRIA, 2011.

[400] Isaac Sheff, Tom Magrino, Jed Liu, Andrew C Myers, and Robbert Van Renesse. Safe serializable secure scheduling: Transactions and the trade-off between security and consistency. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 229–241, 2016.

[401] Isaac Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C Myers. Heterogeneous paxos. In *OPODIS: International Conference on Principles of Distributed Systems*, number 2020 in OPODIS, 2021.

[402] Isaac C Sheff, Robbert van Renesse, and Andrew C Myers. Distributed protocols and heterogeneous trust: Technical report. *arXiv preprint arXiv:1412.3136*, 2014.

[403] Alexander Shraer, Benjamin Reed, Dahlia Malkhi, and Flavio P Junqueira. Dynamic {Reconfiguration} of {Primary/Backup} clusters. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 425–437, 2012.

[404] A Sinha, S. Malik, Chao Wang, and A Gupta. Predictive analysis for detecting serializability violations through trace segmentation. In *MEMOCODE*, 2011.

[405] KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, 2015. ACM.

[406] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 355–364, 1998.

[407] Alberto Sonnino, Shehar Bano, Mustafa Al-Bassam, and George Danezis. Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 294–308. IEEE, 2020.

[408] Francesco Sorrentino, Azadeh Farzan, and P. Madhusudan. Penelope: Weaving threads to expose atomicity violations. In *Proc. FSE*, 2010.

[409] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[410] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.

[411] Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput bft for blockchains. *arXiv preprint arXiv:1906.05552*, page 92, 2019.

[412] Philippe Suter, Robin Steiger, and Viktor Kuncak. Sets with cardinality constraints in satisfiability modulo theories. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 403–418. Springer, 2011.

[413] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proc. SOSP*, 1995.

[414] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 309–324, New York, NY, USA, 2013. ACM.

[415] Robert H Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems (TODS)*, 4(2):180–209, 1979.

[416] Muoi Tran, Inho Choi, Gi Jun Moon, Anh V Vu, and Min Suk Kang. A stealthier partitioning attack against bitcoin peer-to-peer network. In *2020 IEEE symposium on security and privacy (SP)*, pages 894–909. IEEE, 2020.

[417] Nguyen Tran, Jinyang Li, Lakshminarayanan Subramanian, and Sherman SM Chow. Optimal sybil-resilient node admission control. In *2011 Proceedings IEEE INFOCOM*, pages 3218–3226. IEEE, 2011.

[418] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A New Algorithm for Generating All the Maximal Independent Sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[419] Klaus v. Gleissenthall, Rami Gökhan Kici, Alexander Bakst, Deian Stefan, and Ranjit Jhala. Pretend synchrony: synchronous verification of asynchronous distributed programs. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.

[420] Raja Vallée-Rai, Etienne Gagnon, Laurie Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. Optimizing java bytecode using the soot framework: Is it feasible? In DavidA. Watt, editor, *Compiler Construction*, volume 1781 of *LNCS*, pages 18–34. Springer Berlin Heidelberg, 2000.

[421] Marco Vassena, Alejandro Russo, Deepak Garg, Vineet Rajani, and Deian Stefan. From fine-to coarse-grained dynamic information flow control and back. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–31, 2019.

[422] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, Jan 2013.

[423] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

[424] K Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 173–186, 2009.

[425] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6), 2008.

[426] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4(2-3):167–187, 1996.

[427] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, Swiss Federal Institute of Technology, Zurich, 2004.

[428] Anton Wahrstätter, Jens Ernstberger, Aviv Yaish, Liyi Zhou, Kaihua Qin, Taro Tsuchiya, Sebastian Steinhorst, Davor Svetinovic, Nicolas Christin, Mikolaj Barczentewicz, et al. Blockchain censorship. In *Proceedings of the ACM on Web Conference 2024*, pages 1632–1643, 2024.

[429] Liqiang Wang and Scott D. Stoller. Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proc. PPoPP*, 2006.

[430] Liqiang Wang and Scott D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, February 2006.

[431] Xin Wang, Haochen Wang, Haibin Zhang, and Sisi Duan. Pando: Extremely scalable bft based on committee sampling. *Cryptology ePrint Archive*, 2024.

[432] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: Separating data and metadata for efficient and available storage replication. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[433] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed system. In *Proc. PLDI*, 2015.

[434] Levin N Winter, Florena Buse, Daan De Graaf, Klaus Von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized testing of byzantine fault tolerant algorithms. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):757–788, 2023.

[435] Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

[436] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 123–138, New York, NY, USA, 2011. ACM.

[437] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical bft execution. In *Proceedings of the sixth conference on Computer systems*, pages 123–138, 2011.

[438] Zhuolun Xiang, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Strengthened fault tolerance in byzantine fault tolerant replication. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 205–215. IEEE, 2021.

[439] Min Xu, Rastislav Bod´ık, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *Proc. PLDI*, 2005.

[440] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proc. NSDI*, 2009.

[441] 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 *Proc. NSDI*, 2009.

[442] Y. C. Yeh. Safety critical avionics for the 777 primary flight controls system. In *Digital Avionics Systems, 2001. DASC. 20th Conference*, volume 1, pages 1C2/1–1C2/11 vol.1, Oct 2001.

[443] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. Sidetrack: Generalizing dynamic atomicity analysis. In *Proc. PADTAD*, 2009.

[444] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 253–267, New York, NY, USA, 2003. ACM.

[445] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

[446] Haifeng Yu and Amin Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation-Volume 4*, page 21. USENIX Association, 2000.

[447] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

[448] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proc. OSDI*, 2014.

[449] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: A fast blockchain protocol via full sharding. *IACR Cryptol. ePrint Arch.*, 2018:460, 2018.

[450] Pamela Zave. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, March 2012.

[451] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C Myers. Secure program partitioning. *ACM Transactions on Computer Systems (TOCS)*, 20(3):283–328, 2002.

[452] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of CRDTs. volume 8461 of *LNCS*. 2014.

[453] Haibin Zhang and Sisi Duan. Pace: Fully parallelizable bft from reproposable byzantine agreement. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3151–3164, 2022.

[454] Xiaokuan Zhang, Haizhong Zheng, Xiaolong Li, Suguo Du, and Haojin Zhu. You are where you have been: Sybil detection via geo-location analysis in osns. In *2014 IEEE Global Communications Conference*, pages 698–703. IEEE, 2014.

[455] Lantian Zheng, Stephen Chong, Andrew C Myers, and Steve Zdancewic. Using replication and partitioning to build secure distributed systems. In *2003 Symposium on Security and Privacy, 2003.*, pages 236–250. IEEE, 2003.

[456] Lantian Zheng and Andrew C Myers. End-to-end availability policies and noninterference. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 272–286. IEEE, 2005.

[457] Lantian Zheng and Andrew C Myers. A language-based approach to secure quorum replication. In *Proceedings of the Ninth Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 27–39, 2014.