EPFL

Thèse n° 9000

Planetary-Scale Byzantine Fault Tolerance

Présentée le 12 janvier 2024

Faculté informatique et communications
Laboratoire de calcul distribué
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Matteo MONTI

Acceptée sur proposition du jury

Prof. K. Aberer, président du jury
Prof. R. Guerraoui, directeur de thèse
Prof. L. Alvisi, rapporteur
Prof. R. Van Renesse, rapporteur
Prof. B. Ford, rapporteur

École
polytechnique
fédérale
de Lausanne

2024

A Irene.

Preface

This thesis contains selected results of research performed during doctoral studies between September 2017 and August 2023, supervised by Prof. Rachid Guerraoui, at the Distributed Computing Laboratory (School of Computer and Communication Sciences) at EPFL in Lausanne, Switzerland. The results presented in this thesis appear in the following works:

- Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Dragos-Adrian Seredinschi. “Scalable Byzantine Reliable Broadcast". *DISC 2019.* (Best Paper Award.)
- Martina Camaioni, Rachid Guerraoui, Matteo Monti, Manuel Vidigueira. “Oracular Byzantine Reliable Broadcast". *DISC 2022.*
- Martina Camaioni, Rachid Guerraoui, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, Gauthier Voron. “Chop Chop: Byzantine Atomic Broadcast to the Network Limit". *Under Revise-and-Resubmit for OSDI 2024.*

The following additional publications contain work that is not presented in this thesis, but are also a result of research performed during the same doctoral studies:

- Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Dragos-Adrian Seredinschi. “The Consensus Number of a Cryptocurrency". *PODC 2019.*
- Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, Athanasios Xygkis. “Online Payments by Merely Broadcasting Messages". *DSN 2020.* (Runner-up for Best Paper Award.)
- Andrei Kucharavy, Matteo Monti, Rachid Guerraoui, Ljiljana Dolamic. “Byzantine-Resilient Learning Beyond Gradients: Distributing Evolutionary Search." *GECCO 2023.*
- Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, Matteo Monti and Manuel Vidigueira. “Every Bit Counts in Consensus." *DISC 2023.* (Best Paper Award.)
- Martina Camaioni, Rachid Guerraoui, Jovan Komatovic, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, Gauthier Voron. “Carbon: Scaling Trusted Payments with Untrusted Machines." *Under review for TDSC 2023.*

The following additional publications contain results from prior work in physics, but published during the same doctoral studies:

- Armando Bazzani, Silvia Vitali, Carlo Emilio Montanari, Matteo Monti, Gastone Castellani. “Sthocastic Properties of Colliding Particles in a Non-Equilibrium Thermal Bath." *Nonlocal and Fractional Operators 2019*.
- Silvia Vitali, Carlo Emilio Montanari, Matteo Monti, Gastone Castellani. “Event-Based Simulation of 2D Particles Gas in a Gradient of Temperature." *ICTT 2019*.

Abstract

The scale and pervasiveness of the Internet make it a pillar of planetary communication, industry and economy, as well as a fundamental medium for public discourse and democratic engagement. In stark contrast with the Internet's decentralized infrastructure, however, most hyper-scale online services adopt a centralized architecture, wherein millions or billions of clients entrust the same service provider with sensitive data to store, process and transmit, resulting in poor security and a dangerous asymmetry of power between users and system maintainer. Byzantine distributed computing has long held the promise to advance the security and transparency of our information infrastructure, replicating services across several, mutually mistrustful processes so as to uphold security and transparency even if a fraction of the processes deviate arbitrarily from the algorithm they are assigned. Despite extensive research, however, real-world replicated systems still fall short of global adoption. This is at least in part due to the limited scalability of Byzantine algorithms: the cost of replication is still too high, preventing the rise of Byzantine hyper-scalers.

This thesis attempts to overcome this limitation, contributing planetary-scale implementations for two classes of distributed abstractions: Reliable Broadcast, which has correct processes agree on messages issued by individual sources, and Consensus, which provides agreement on values contributed by multiple independent sources. The two classes of abstraction differ in power and requirements: Consensus is more powerful, enabling universal distributed computation, but Reliable Broadcast can be implemented without any assumption on network timeliness, a better fit for the public Internet's large latency spikes.

First, we scale the number of servers that can take part in Reliable Broadcast. We present Contagion, the first probabilistic Reliable Broadcast protocol to achieve logarithmic per-process computation and communication complexity, enabling scalability to a practically unlimited number of servers. At the core of Contagion are samples, a probabilistic alternative to Byzantine quorums foregoing intersection guarantees for statistical representativeness. Contagion's security evaluation is among the main technical contributions of this thesis, providing the first formal analysis of a probabilistic protocol in the Byzantine setting.

Second, we scale the number of clients that can concurrently submit messages to a Reliable Broadcast system. We present Draft, the first implementation of Reliable Broadcast to asymptotically amortize (whenever a set of commonplace conditions are met) all signature and communication overhead resulting from Byzantine fault tolerance, matching the complexity of a trusted, centralized solution. Draft's efficiency is enabled by brokers, a novel

layer of untrusted processes we introduce between clients and servers to enhance system performance.

Finally, we scale the number of clients a Consensus-class system can handle. We do so with Chop Chop, the first system to introduce brokers to the Atomic Broadcast abstraction. When geo-deployed on 64 medium-sized servers, Chop Chop processes 43,600,000 messages per second, two orders of magnitude more than state-of-the-art alternatives. Even at maximum load, Chop Chop's performance is close to line-rate, putting 92% of server bandwidth towards transmitting application messages, negating nearly all cost resulting from Byzantine fault tolerance.

Keywords. Distributed systems, Byzantine fault tolerance, scalability, planetary scale, replication, reliable broadcast, consensus, randomized algorithms, broker-enhanced computation, oracularity, efficiency.

Résumé

L'ampleur et l'omniprésence d'Internet en font un pilier de la communication, de l'industrie et de l'économie à l'échelle planétaire, ainsi qu'un support fondamental pour le discours public et l'engagement démocratique. Cependant, contrairement à l'infrastructure décentralisée d'Internet, la plupart des services en ligne à grande échelle adoptent une architecture centralisée, dans laquelle des millions ou des milliards de clients confient au même fournisseur de service des données sensibles à stocker, à traiter et à transmettre, ce qui entraîne une sécurité médiocre et une dangereuse asymétrie de pouvoir entre les utilisateurs et le responsable du système. Le calcul distribuée Byzantin promet depuis longtemps de faire progresser la sécurité et la transparence de notre infrastructure d'information, en répliquant les services à travers plusieurs ordinateurs qui se méfient les uns des autres, de manière à maintenir la sécurité et la transparence même si une fraction des ordinateurs dévient arbitrairement de l'algorithme qui leur est assigné. Malgré des recherches approfondies, les systèmes répliqués n'ont pas encore été adoptés en pratique à l'échelle mondiale. Cela est dû, au moins en partie, a la capacité limitée pour des algorithmes Byzantins de passer à l'échelle : le coût de la réplication est encore trop élevé, ce qui empêche l'apparition d'*hyper-scalers* Byzantins.

Cette thèse tente de surmonter cette limitation par des implémentations, utilisables à l'échelle planétaire, pour deux classes d'abstractions distribuées : le *Reliable Broadcast*, qui permet aux processus corrects de se mettre d'accord sur des messages émis par des sources individuelles, et le *Consensus*, qui permet de se mettre d'accord sur des valeurs fournies par de multiples sources indépendantes. Les deux catégories d'abstractions diffèrent en termes de puissance et d'exigences : le *Consensus* est plus puissant et permet un calcul distribué universel, mais le *Reliable Broadcast* peut être mis en œuvre sans aucune hypothèse sur la latence du réseau, ce qui est mieux adapté aux importants pics de latence de l'Internet public.

Tout d'abord, nous augmentons le nombre de serveurs pouvant participer aux *Reliable Broadcast*. Nous présentons Contagion, le premier protocole probabiliste de *Reliable Broadcast* dont la complexité de calcul et de communication par processus est logarithmique, ce qui permet de l'étendre à un nombre pratiquement illimité de serveurs. Au cœur de Contagion se trouvent les échantillons, une alternative probabiliste aux quorums Byzantins qui renoncent aux garanties d'intersection pour la représentativité statistique. L'évaluation de la sécurité de Contagion est l'une des principales contributions techniques de cette thèse, car elle fournit la première analyse formelle d'un protocole probabiliste dans le cadre Byzantin.

Ensuite, nous augmentons le nombre de clients qui peuvent simultanément soumettre des

messages à un système de *Reliable Broadcast*. Nous présentons Draft, la première implémentation de *Reliable Broadcast* à amortir asymptotiquement, chaque fois qu'une série de conditions communes sont satisfaites, l'intergralité du surcoût des signatures et des communications résultant de la tolérance aux pannes Byzantines, et ainsi atteindre la complexité d'une solution centralisée et fiable. L'efficacité de Draft est rendue possible par les *brokers*, une nouvelle classe de processus ne nécessitant pas de garantie de fiabilité que nous introduisons entre les clients et les serveurs pour améliorer les performances du système.

Enfin, nous augmentons le nombre de clients qu'un système de type *Consensus* peut gérer. Nous le faisons avec Chop Chop, le premier système à introduire des *brokers* dans l'abstraction *Atomic Broadcast*. Lorsqu'il est géodéployé sur 64 serveurs de taille moyenne, Chop Chop traite 43 600 000 messages par seconde, soit deux ordres de grandeur de plus que l'état de l'art. Même à la charge maximale, les performances de Chop Chop sont proches du *line-rate*, consacrant 92% de la bande passante du serveur à la transmission des messages d'application, ce qui annule la quasi-totalité des coûts résultant de la tolérance aux pannes Byzantines.

Contents

Preface i

Abstract (English / Français) iii

Introduction 1

I Scalable Byzantine Reliable Broadcast 9

1 Overview 11

1.1 Introduction . . . 11

1.1.1 Probabilistic Byzantine Reliable Broadcast . . . 12

1.1.2 Probability Analysis . . . 13

1.1.3 Security and Complexity Evaluation . . . 14

1.2 Related Work . . . 16

1.3 Model . . . 17

2 Murmur 19

2.1 Interface . . . 19

2.2 Algorithm . . . 20

2.3 No duplication, integrity and validity . . . 20

2.4 Totality . . . 22

3 Sieve 27

3.1 Interface . . . 27

3.2 Algorithm . . . 28

3.3 No duplication and integrity . . . 30

3.4 Total validity . . . 31

3.5 Preliminary lemmas . . . 32

3.6 Simplified Sieve . . . 38

3.6.1 Consistency-only broadcast . . . 38

3.6.2 Byzantine oracle . . . 39

3.6.3 Algorithm . . . 39

3.7 Adversarial execution . . . 44

3.7.1 Model (Sieve) . . . 44

3.7.2 Model (Simplified Sieve) . . . 45
3.7.3 Network scheduling . . . 45
3.7.4 Interfaces . . . 46
3.8 Simplified adversarial power . . . 48
3.8.1 Preliminary definitions . . . 48
3.8.2 Consistency of Simplified Sieve . . . 50
3.9 Two-phase adversaries . . . 58
3.10 Consistency . . . 66
3.10.1 Two-phase adversaries . . . 67
3.10.2 Random variables . . . 68
3.10.3 Byzantine population, correct echoes, delivery . . . 70
3.10.4 Second phase . . . 72
3.10.5 First phase . . . 78
3.11 Decorators . . . 82
3.11.1 Auto-echo adversary . . . 82
3.11.2 Process-sequential adversary . . . 88
3.11.3 Sequential adversary . . . 97
3.11.4 Non-redundant adversary . . . 108
3.11.5 Sample-blind adversary . . . 114
3.11.6 Byzantine-counting adversary . . . 127
3.11.7 Single-response adversary . . . 133
3.11.8 Two-phase adversary . . . 139

4 Contagion **147**
4.1 Interface . . . 147
4.2 Algorithm . . . 148
4.3 No duplication and integrity . . . 150
4.4 Validity . . . 151
4.5 Adversarial execution . . . 152
4.5.1 Model . . . 152
4.6 Epidemic processes . . . 153
4.7 Threshold contagion (overview) . . . 157
4.8 Preliminary lemmas . . . 159
4.9 Consistency . . . 161
4.10 Totality . . . 163
4.10.1 Minimal operations . . . 163
4.10.2 Delivery probability . . . 164
4.10.3 C-step Threshold Contagion . . . 165
4.11 Threshold Contagion . . . 167
4.11.1 Epidemic processes . . . 168
4.11.2 Threshold Contagion . . . 169
4.11.3 Rules . . . 169

4.11.4 Random variables . . . 172
4.11.5 Goal . . . 172
4.11.6 Sample space . . . 173
4.11.7 Random variables as sample functions . . . 178
4.11.8 Contagion step . . . 182
4.11.9 Final infection size . . . 190

II Oracular Byzantine Reliable Broadcast **193**

5 Overview **195**
5.1 Introduction . . . 195
5.2 Related Work . . . 201
5.3 Model & background . . . 203
5.3.1 Model . . . 203
5.3.2 Background . . . 204

6 Draft **205**
6.1 Interface . . . 205
6.2 Algorithm . . . 206
6.2.1 Protocol & correctness overview . . . 206
6.2.2 Complexity overview . . . 212
6.2.3 Pseudocode (Client) . . . 214
6.2.4 Pseudocode (Broker) . . . 216
6.2.5 Pseudocode (Server) . . . 221
6.3 Correctness . . . 227
6.3.1 No duplication . . . 227
6.3.2 Consistency . . . 227
6.3.3 Totality . . . 232
6.3.4 Integrity . . . 234
6.3.5 Validity . . . 236
6.4 Complexity . . . 252
6.4.1 Auxiliary results . . . 252
6.4.2 Batching limit . . . 260
6.4.3 Protocol analysis . . . 261

7 Dibs **267**
7.1 Interface . . . 267
7.2 Algorithm . . . 269
7.2.1 Pseudocode (Client) . . . 269
7.2.2 Pseudocode (Server) . . . 271
7.3 Correctness . . . 272
7.3.1 Correctness . . . 272
7.3.2 Signup integrity . . . 275

7.3.3 Signup validity . . . 275
7.3.4 Self-knowledge . . . 277
7.3.5 Transferability . . . 278
7.3.6 Density . . . 279

III Byzantine Atomic Broadcast to the Network Limit 281

8 Chop Chop 283
8.1 Overview . . . 283
8.2 Atomic Broadcast . . . 286
8.2.1 Cost of Atomic Broadcast . . . 286
8.2.2 Existing Mitigations . . . 287
8.3 Distilled Batches . . . 288
8.3.1 Distillation at a Glance . . . 289
8.3.2 Distillation Microbenchmark . . . 290
8.4 Chop Chop . . . 291
8.4.1 Architecture and Model . . . 292
8.4.2 Distillation Phase . . . 293
8.4.3 Submission Phase . . . 296
8.4.4 Correctness . . . 297
8.5 Implementation Details . . . 299
8.5.1 Broker . . . 299
8.5.2 Server . . . 300
8.6 Evaluation . . . 301
8.6.1 Baselines . . . 301
8.6.2 Setup . . . 302
8.6.3 RQ1 – Load Handling . . . 305
8.6.4 RQ2 – Distillation Benefits . . . 306
8.6.5 RQ3 – Number of Servers . . . 308
8.6.6 RQ4 – Overall Efficiency . . . 309
8.6.7 RQ5 – Chop Chop Under Failures . . . 309
8.6.8 RQ6 – Application Use Cases . . . 310
8.7 Related Work . . . 310

Conclusions 313

Bibliography 315

Introduction

A word of motivation

The Internet was designed with decentralization in mind, as a way to maintain reliable communications in the face of catastrophic infrastructural damage. Half a century after ARPANET's first deployment, the scale and pervasiveness of the Internet make it a pillar of planetary communication, industry and economy [int21], as well as a fundamental medium for public discourse and democratic engagement [ESWV22]. In stark contrast with the infrastructure on which they rely, however, most Internet-scale services adopt a centralized architecture, wherein millions or billions of clients entrust the same service provider with sensitive data to store, process and transmit [Fit22, Run20].

Centralized computer systems are fast, scalable and easy to deploy, but come at a severe cost in terms of security and transparency. On the one hand, single points of failure are notoriously bad for security, as even the best-maintained systems are prone to being hacked or compromised by random accidents. In 2023, cyber-attacks are projected to account for as much as 8 trillion US dollars in global economic damage [Far22] - over 8% of the world's economy, lost to improperly secured computer systems. In the last year alone, real-world attacks have been successfully dealt [cyb23] on energy systems, financial infrastructure, healthcare providers, governments and military organizations alike, both in war and peacetime. However well maintained, a centralized system is prone to being hacked into, physically tampered with, or compromised by random accidents.

On the other hand, centralized solutions force users to trust that the system's maintainer (often a self-interested, profit-driven organization) will behave in a way aligned with the users' best interests, e.g., by preserving data confidentiality or upholding accurate bookkeeping even when failing to do so would be advantageous and difficult to detect. Over the last decade, this power asymmetry resulted in a long sequence of abuses, with Internet-scale service providers violating user privacy [Reu20, Cer19, Pri19], mismanaging hate speech [Moz18], enabling democratic tampering [Con18] and harming the health of their most vulnerable users [Gay21]. Lacking ad-hoc transparency measures, centralized systems are opaque. Once cleartext data is handed off, it is not easy to determine how that data will be handled.

Byzantine distributed computing, on which this thesis focuses, has long [1] held the promise to advance the security and transparency of humanity's information infrastructure. By replicating a service across several, reciprocally mistrusting *processes*, a Byzantine-resilient system upholds well-defined liveness and safety properties even if a fraction of the processes fail maliciously, deviating arbitrarily from the algorithm they are are assigned. In theory, any service can be replicated, faithfully and transparently executed on a common system whose correctness and availability would be very hard to thwart. As such, Byzantine algorithms hold the promise of profound societal impact, securing critical infrastructure and empowering citizens with services whose operation is auditable and democratically established, beyond the influence of any individual party.

Despite extensive research and interest from industry and governments, however, real-world Byzantine systems still fall short of global adoption: from search engines to social networks, from messaging platforms to software distribution infrastructures, most planetary-scale services still adopt, at the time of writing, a centralized architecture. This lack of adoption, we argue, is at least in part[2] due to the limited scalability of Byzantine algorithms: the cost of replication is still too high, preventing the rise of Byzantine hyper-scalers.

The Internet's foundational vision was that of a highly-available, decentralized, secure, universal computer shared by all. This thesis hopes to make a small step towards that vision, introducing practical distributed algorithms that billions of stakeholders can partake in as servers, or use as clients.

Two levels of power, two ways to scale

This thesis focuses on two fundamental classes of distributed abstraction: Reliable Broadcast and Consensus. Abstractions within the same class can be implemented from each other (for example, Atomic Broadcast can be built atop Consensus, and vice versa). While Consensus can be used to implement Reliable Broadcast, however, the converse is provably impossible [68]. As such, abstractions in the Consensus class are strictly more powerful than those in the Reliable Broadcast class. Consensus-powered algorithms, however, are generally more complex, and require more stringent assumptions about the system on which they run.

Reliable Broadcast. Abstractions in the Reliable Broadcast class have correct processes agree on messages issued by individual sources. Reliable Broadcast [31], the class' namesake, has one designated sender broadcast a single message. If the sender is correct, every correct process delivers the message it broadcasts. Even if the sender is Byzantine, all correct processes

[1] The seminal *"Byzantine Generals Problem"* [104] was published in 1982, over forty years prior to the writing of this thesis.

[2] Other factors might include the high latency inherent to WAN-based coordination, the lack of well-established, practical solutions for deploying real-world distributed systems, and the incomplete legislative frameworks regulating decentralised networks. This thesis aims to contribute technical results to the public discourse, but the road ahead is likely long and interdisciplinary.

deliver either the same message, or no message at all. Multi-shot variants [34] of Reliable Broadcast include Source-Order Broadcast, wherein every process maintains an independent, agreed-upon, append-only log, and Causal Broadcast, wherein no message is delivered before any of its causal antecedents. Reliable Broadcast has well-known applications [37, 61] as a building block in practical fault-tolerant systems. Most notably, the Asset-Transfer problem, which cryptocurrency-based systems traditionally solve using Consensus, was recently reduced [80] to Causal Broadcast, showing Reliable Broadcast's relevance as a stand-alone solution. Abstractions in the Reliable Broadcast class are generally considered simpler to implement [50] than their Consensus-based counterparts. Maybe more importantly, Reliable Broadcast can be solved in the asynchronous setting [31]: all abstractions in the Reliable Broadcast class uphold liveness and safety even when no assumption can be made on how rapidly processes can exchange messages. This makes Reliable Broadcast solutions fast and secure in environments with large latency spikes, such as the public Internet. Simplicity and asynchrony, however, come at the cost of generality, as only a relatively narrow class of problems [3] can be solved by abstractions in the Reliable Broadcast class - Consensus is still required in the general case. We study abstractions in the Reliable Broadcast class in Parts I and II of this thesis.

Consensus. Abstractions in the Consensus class have correct processes agree on values contributed by multiple independent sources. Consensus [104], the class' namesake, has each correct process first propose, then decide a value. All correct processes eventually decide the same value; if all correct processes propose the same value[4], that value must be decided. The canonical multi-shot variant of Consensus is Atomic Broadcast, wherein all sources contribute messages to the same, totally-ordered, agreed-upon, append-only log. The most relevant application of Atomic Broadcast is arguably State Machine Replication, the fundamental distributed abstraction enabling universal Byzantine-resilient computation. In brief, State Machine Replication uses Atomic Broadcast to order requests, usually contributed by an external set of clients. Every correct process initializes a local copy of the same deterministic state matchine to the same initial value, then serves the same requests in the same order. Because all correct processes deterministically transition through the same sequence of states, any response backed by sufficiently many processes is guaranteed to be correct - this technique can be employed to make any deterministic service safe and available despite a (suitably small) fraction of processes misbehaving arbitrarily. Abstractions in the Consensus class usually call for more complex implementations than their Reliable Broadcast counterparts. Additionally, Consensus is provably impossible in asynchrony [68][5], forcing Consensus algorithms to make assumptions about the timeliness of inter-process communication. We study abstractions in

[3] Reliable Broadcast can only solve problems whose *consensus number* is 1 Herlihy's hyerarchy [86].

[4] Several other flavors of *validity* are defined in literature [48], relating decisions to proposals. Weaker forms of validity, for example, only apply if all processes are correct.

[5] The impossibility of asynchronous Consensus applies only to the deterministic variant of Consensus. Probabilistic variants of Consensus, which deterministically uphold safety but only terminate with probability 1, do allow for asynchronous solutions.

the Consensus class in Part III.

The goal of this thesis is to contribute planetary-scale implementations for both classes of abstraction. For each class, we attempt in particular to scale the number of processes that can partake in the abstraction as *servers*, as well as the number of processes that can concurrently use the abstraction as *clients*.

Scaling servers. Designing a distributed algorithm that can scale to billions of servers presents some fundamental challenges. At the core of even the simplest distributed algorithms [31] is the concept of *quorums* [149], sets of servers that are large enough to intersect in at least one correct server. Classically, deterministic algorithms [22, 31, 43, 156] have at least one process exchange messages with a quorum of servers, resulting in an unworkable communication complexity: at planetary scale, no individual process can feasibly communicate with (a non-negligible fraction of) the whole system. We tackle this fundamental problem by shifting to the probabilistic setting, replacing quorums with **samples**. Unlike quorums, samples do not provide any intersection guarantees - a sample is just large enough to be (with high probability) statistically representative of the system as a whole. Studying the security of a probabilistic system while accounting for Byzantine behavior is one of the main challenges of this thesis. We focus on server scalability in Part I of this thesis.

Scaling clients. In order to reliably service billions of clients, a system will realistically need to sustain millions of requests per second. Designing a distributed algorithm for such a high throughput requires minimizing the amount of computation and communication required server-side to process each request. Techniques such as *batching* [43, 55, 70, 140] are available in literature to partially amortize the cost of coordination among servers. Even when batching is employed, however, the communication and computation cost incurred by a server when processing a high rate of requests is often dominated by operations (such as message authentication) that, we notice, are expensive to *perform*, but cheap to *verify*. To leverage this observation, we extend the classical client-server model with **brokers**, a permissionless, scalable set of processes whose only purpose is to alleviate server complexity. Unlike servers, a large fraction of which we must assume to be correct, all brokers but one can be faulty. Brokers act as an intermediary between clients and servers, taking upon themselves to perform expensive, verifiable operations. We focus on client scalability in Parts II and III of this thesis.

Contributions of this thesis

This thesis (Figure 1) presents solutions to three of the four scalability challenges we outlined above: we scale Reliable Broadcast servers in Part I, Reliable Broadcast clients in Part II, and Consensus clients in Part III.

	Reliable Broadcast		Consensus	
	Theory	*Systems*	*Theory*	*Systems*
Server Scalability	Part I, [81]	*(unpublished)*		
Client Scalability	Part II, [41]	[39] *(submitted)*	*(work in progress)*	Part III, [40] *(minor revisions)*

Figure 1: Contributions of this thesis.

Part I: Scalable Byzantine Reliable Broadcast. We generalize single-shot Reliable Broadcast to the probabilistic setting, allowing each of its properties to be violated with a fixed, arbitrarily small probability. We leverage these relaxed guarantees in a protocol replacing quorums with stochastic samples. Unlike quorums, samples do not provide any intersection guarantee: a sample is just large enough to be statistically representative of the overall system (at least with high probability). We implement Reliable Broadcast with three algorithms, building on top of each other to progressively obtain stronger abstractions: Murmur for Probabilistic Broadcast, Sieve for Probabilistic Consistent Broadcast, and Contagion for Probabilistic Reliable Broadcast. To the best of our knowledge, Contagion is the first Reliable Broadcast protocol to achieve logarithmic per-process computation and communication complexity, enabling scalability to a practically unlimited number of servers. A major technical contribution of Part I is a fully formal analysis of our protocol, deriving closed-form bounds on the probability of each Reliable Broadcast property being compromised. The complexity of such an analysis is exacerbated in the Byzantine setting, as each bound must hold for every possible adversarial strategy. We tackle this problem using decorators, a novel suite of formal techniques designed to progressively narrow down the set of strategies that provably includes the optimal adversary.

The algorithms and techniques discussed in Part I were presented [81] at the International Symposium on Distributed Computing (DISC 2019). The paper won the Best Paper Award for that edition of the conference. In addition to the theoretical results presented in this thesis, a C++ implementation of Contagion was developed and tested on a global deployment of Amazon AWS machines. Preliminary results showed a throughput of upwards of a thousand messages per second even when Contagion was deployed on 2048 Micro instances (larger-scale testing was unfeasbile due to budget constraints). Regrettably, the experimental results on Contagion are, at the time of writing, unpublished.

Part II: Oracular Byzantine Reliable Broadcast. We study Client-Server Byzantine Reliable Broadcast, a multi-shot variant of Reliable Broadcast whose interface is split between broadcasting clients and delivering servers. We present Draft, an optimally resilient implementation of Client-Server Byzantine Reliable Broadcast. Like most algorithms in the Reliable Broadcast class, Draft guarantees both liveness and safety in the asynchronous setting. Under good conditions, however, Draft stretches towards information-theoretical efficiency. In a moment

of synchrony, free from Byzantine misbehavior, and at the limit of infinitely many broadcasting clients, a Draft server delivers a b-bits payload at an asymptotic amortized cost of 0 signature verifications, and $\log_2(c) + b$ bits exchanged, where c is the number of clients in the system. This is the minimum number of bits required to convey the payload (b bits, assuming it is incompressible) along with an identifier for its sender ($\log_2(c)$ bits, necessary to enumerate any set of c elements, and optimal if broadcasting frequencies are uniform or unknown). Noting that this is the same complexity a server would incur by receiving plain, unauthenticated messages from a trusted oracle, we say that Draft can achieve *oracularity*, effectively achieving zero-cost Byzantine fault tolerance. We achieve oracularity by noting that a synchronous, interactive protocol can, in the absence of Byzantine behavior, reduce the metadata of a batch of messages down to a constant. Remarkably, this reduction protocol is expensive to perform, but publicly verifiable: ill-reduced batches are visibly malformed, and can be discarded without additional coordination. This observation allows us to offload reduction to brokers, a novel set of processes we add to the distributed computing model between clients and servers. Brokers play no safety role in the system (even if they all fail, only[6] liveness is lost), and can be trustlessly spun up to meet client demand, interacting with clients to produce reduced batches for the servers to oracularly process.

The results discussed in Part II were presented [41] at the International Symposium on Distributed Computing (DISC 2022). In addition to the theoretical results presented in this thesis, Draft was generalized into Carbon, a reconfigurable, garbage-collected, asynchronous asset-transfer system. When tested on a global deployment of 64 AWS machines, a Rust implementation of Carbon sustained upwards of one million messages per second. Our experimental results are currently under review for publication [39] in the IEEE Transactions on Dependable and Secure Computing (TDSC).

Part III: Byzantine Atomic Broadcast to the Network Limit. We present Chop Chop, a Byzantine Atomic Broadcast system that amortizes the cost of ordering, authenticating and deduplicating messages, achieving *line rate* (i.e., closely matching the complexity of a protocol that does not ensure any ordering, authentication or Byzantine resilience) even when processing messages as small as 8 bytes. Chop Chop attains this performance by means of *distillation*, a generalization of the reduction techniques we introduced in Part II for the Reliable Broadcast class. A distilled batch is a set of messages that are fast to authenticate and deduplicate, as well as order. Batches are distilled using a novel, interactive protocol based on brokers. In a geo-distributed deployment of 64 medium-sized servers, with clients situated cross-cloud, a Rust implementation of Chop Chop processes 43,600,000 messages per second with an average latency of 3.6 seconds. Under the same conditions, state-of-the-art alternatives offer two orders of magnitude less throughput for the same latency. We test three simple applications,

[6] The liveness of a critical system might arguably be as important as its safety. The trustless nature of brokers, however, helps protect liveness too. Should well-established brokers experience catastrophic failure, concerned stakeholders could, e.g., spin up their own brokers, bypassing lengthy security audits to quickly bring the system back online.

running on a replicated state machine powered by Chop Chop: a Payment system, an Auction house and an instance of the "Pixel War" game, respectively achieving 32, 2.3 and 35 million operations per second.

The results presented in Part III are under minor revisions for presentation [40] at the USENIX Symposium on Operating Systems Design and Implementation (OSDI 2024). In addition to the experimental results presented in this thesis, a paper is under development to generalize oracularity to the Atomic Broadcast abstraction, further establish brokers as a well-defined role in the distributed computing model, and prove Chop Chop's correctness and oracularity to the fullest extent of formal detail.

State of the art. To the best of our knowledge, all three results presented in this thesis are state of the art at the time of writing. An implementation of Reliable Broadcast with sub-logarithmic complexity would likely require going past stochastic sampling: even assuming that only the sender is Byzantine, having every correct process query a sub-logarithmic sample would result in a non-negligible probability of non-representativeness for at least one sample, effectively poisoning a process' view of the overall system. Concerning our oracular and line-rate implementations, improving efficiency would necessarily involve some compression scheme: assuming ids and messages are incompressible, both solutions asymptotically reach, to the bit, the information-theoretical lower bound.

Towards Scalable Byzantine Atomic Broadcast. A great deal of effort was put towards the development of a Byzantine Atomic Broadcast algorithm that (similarly to Contagion) would achieve logarithmic per-process computation and communication complexity. Doing so proved unfeasible within the time allowed by PhD research. While an algorithm, which we conjecture to be correct, was developed, a fully formal proof of its safety and liveness proved intractable within such a limited time frame. As we discuss in Part I, much of the complexity of Contagion's analysis stems from the non-trivial interaction between the probabilistic behavior of correct processes and the arbitrary behavior of the Byzantine adversary. Adding to this the subtlety of time analysis (our Atomic Broadcast algorithm works in the partially synchronous model) further complicates our proofs. On the way to proving the algorithm's correctness, a fully formal theory of decorators was developed, framing Byzantine behavior in the general scope of probabilistic games. Due to its length (comparable to the entire extent of this thesis) and dubious standing as stand-alone contribution, our theoretical work on decorators was omitted from this thesis, and is currently unpublished.

Part I

Scalable Byzantine Reliable Broadcast

1 Overview

1.1 Introduction

Broadcast is a popular abstraction in the distributed systems toolbox, allowing a process to transmit messages to a set of processes. The literature defines many flavors of broadcast, with different safety and liveness guarantees [34, 72, 82, 110, 125]. In this Part we focus on Byzantine reliable broadcast, as introduced by Bracha [31]. This abstraction is a central building block in practical Byzantine fault-tolerant (BFT) systems [37, 61, 80]. We tackle the problem of its scalability, namely reducing the complexity of Byzantine reliable broadcast, and seeking good performance despite a large number of participating processes.

In Byzantine reliable broadcast, a designated sender broadcasts a single message. Intuitively, the broadcast abstraction ensures that no two correct processes deliver different messages (*consistency*), either all correct processes deliver a message or none does (*totality*), and that, if the sender is correct, all correct processes eventually deliver the broadcast message (*validity*). This must hold despite a certain fraction of Byzantine processes, potentially including the sender. We denote by N the number of processes in the system, and f the fraction of processes that are Byzantine. Existing algorithms for Byzantine reliable broadcast scale poorly as they typically have $O(N)$ per-process communication complexity [32, 110, 111, 146]. The root cause for the poor scalability of these algorithms is their use of quorums [112, 149], i.e., sets of processes that are large enough to always intersect in at least one correct process. The size of a quorum grows linearly with the size of the system [34].

To overcome the scalability limitation of quorum-based broadcast, Malkhi *et al.* [113] generalized quorums to the probabilistic setting. In this setting, two random quorums intersect with a fixed, arbitrarily high probability, allowing the size of each quorum to be reduced to $O(\sqrt{N})$. We are not aware of any Byzantine reliable broadcast algorithm building on probabilistic quorums; nevertheless, such an algorithm could have a per-process communication complexity reduced from $O(N)$ to $O(\sqrt{N})$. The active$_t$ protocol of Malkhi *et al.* [110] uses a form of samples for an optimistic path, but relies on synchrony and has a linear worst-case complexity (that is arguably very likely to occur with only moderate amounts of faulty processes).

Samples. In this Part, we present a probabilistic gossip-based Byzantine reliable broadcast algorithm having $O(\log N)$ per-process communication and computation complexity, at the expense of $O(\log N / \log\log N)$ latency. Essentially, we propose *samples* as a replacement for quorums. Like a probabilistic quorum, a sample is a randomly selected set of processes. Unlike quorums, samples do not need to intersect. Any two quorums must have a large enough size to ensure their intersection in a correct node (at least with high probability). Samples, instead, can be significantly smaller than quorums, as each sample must be large enough only to be *representative* of the system with high probability.

As with quorums, a process can use its sample to gather information about the global state of the system. An old Italian saying provides an intuitive understanding of this shift of paradigm: *"To know if the sea is salty, one needs not drink all of it!"* Intuitively, we leverage the law of large numbers, trading performance for a fixed, arbitrarily small probability of non-representativeness. To get an intuition of the difference between quorums and samples, consider the emulation of a shared memory in message passing [9]. One writes in a quorum and reads from a quorum to fetch the last value written. Our algorithms are rather in the vein of "write all, read any". Here we would "write" using a gossip primitive and "sample" the system to seek the last value.

Throughout this Part, we extensively use samples to estimate the number of processes satisfying a set of *yes-or-no* predicates, e.g., the number of processes that are ready to deliver a message m. Consider the case where a correct process π queries K randomly selected processes (a sample) for a predicate P. Assume a fraction p of correct processes from the whole system satisfy predicate P. Let x be the fraction of positive responses (out of K) that π collects. By the Chernoff bound, the probability of $\left|x - p\right| \geq f + \epsilon$ is smaller or equal to $\exp(-\lambda(\epsilon)K)$, where λ quickly increases with ϵ. For sufficient K, the probability of x differing from p by more than $f + \epsilon$ can be made exponentially small.

To obtain a sample, our algorithms use a *sampling oracle* that returns the identity of a process from the system picked with uniform probability. In a permissioned system (i.e., one where the set of participating processes is known) sampling reduces to picking with uniform probability an element from the set of processes. In a permissionless system subject to Byzantine failures and slow churn, a (nearly) uniform sampling mechanism is still achievable using gossip [28].

1.1.1 Probabilistic Byzantine Reliable Broadcast

Our probabilistic algorithm, Contagion, allows each property of Byzantine reliable broadcast to be violated with an arbitrarily small probability ϵ. We show that ϵ scales sub-quadratically with N, and decays exponentially in the size of the samples. As a result, for a fixed value of ϵ, the per-node communication complexity of Contagion is logarithmic.

We build Contagion incrementally, relying on two sub-protocols, as we describe next.

First, **Murmur** is a probabilistic broadcast algorithm that uses simple message dissemination

to establish *validity* and *totality*. In this algorithm, each correct process relays the sender's message to a randomly picked *gossip sample* of other processes. For the sample size $\Omega(\log N)$, the resulting gossip network is a connected graph with $O(\log N / \log\log N)$ diameter, with high probability [47, 65]. In case of a Byzantine sender, however, Murmur does not guarantee consistency.

Second, **Sieve** is a probabilistic consistent broadcast algorithm that guarantees *consistency*, i.e., no two correct processes deliver different messages. To do so, each correct process uses a randomly selected *echo sample*. Intuitively, if enough processes from any echo sample confirm a message m, then with high probability no correct processes in the system delivers a different message m'. Sieve, however, does not ensure totality. If a Byzantine sender broadcasts multiple conflicting messages, a correct process might be unable to gather sufficient confirmations for either of them from its echo sample, and consequently would not deliver any message, even if some correct process delivers a message.

Finally, **Contagion** is a probabilistic reliable broadcast algorithm that guarantees validity, consistency, and totality. The sender uses Sieve to disseminate a consistent message to a subset of the correct processes. In order to achieve totality, Contagion mimics the spreading of a contagious disease in a population. A process samples the system and if it observes enough other "infected" processes in its sample, it becomes infected itself. If a critical fraction of processes is initially infected by having received a message from the underlying Sieve layer, the message spreads to all correct processes with high probability. If a process observes enough other infected processes, it delivers. As in the original deterministic implementation by Bracha [31], the crucial point here is that "enough" for becoming infected is less than "enough" for delivering. This way, with high probability, either all correct processes deliver a message or none does—Contagion satisfies totality. The other two important properties (validity and consistency) are inherited from the underlying (Murmur and Sieve) layers.

1.1.2 Probability Analysis

A major technical contribution of this Part is a complete, formal analysis of the properties of our three algorithms. To the best of our knowledge, this is the first analysis of a probabilistic broadcast algorithm in the Byzantine fault model, and this turned out to be very challenging. Intuitively, providing a bound on the probability of a property being violated reduces to studying a joint distribution between the inherent randomness of the system and the behavior of the Byzantine adversary. Since the behavior of the adversary is arbitrary, the marginal distribution of the Byzantine's behavior is unknown.

We develop two novel strategies to bound the probability of a property being violated, which we use in the analysis of Sieve and Contagion respectively.

Identifying the optimal adversarial strategy. When evaluating the consistency of Sieve, we show that a bound holds for every *possibly optimal* adversarial strategy. Essentially, we identify a subset of adversarial strategies that we prove to include the optimal one, i.e., the one that has the highest probability of compromising Sieve's consistency. We then prove that every possibly optimal adversarial strategy has a probability of compromising the consistency of Sieve smaller than some ϵ.

Identifying a set of possibly optimal adversarial strategies that is narrow enough to be amenable to analytic bounds is the most technically involved challenge of this Part. We do so by means of a novel technique that involves modeling the adversary as an algorithm interacting with the system through a well-defined interface. We start from the set of all possible adversarial algorithms (trivially, this set includes an optimal adversary). We then reduce the set of possibly optimal adversaries in steps. At every step, we transform each adversary by means of a *decorator*. Intuitively, a decorator tweaks an adversary in a way that simplifies its behavior without reducing its effectiveness in compromising the system. With each decorator, the set of possibly optimal adversaries becomes narrower and more tractable, finally yielding adversaries whose behavior is so simple that the probability of consistency being compromised can be computed analytically.

Making adversarial strategy irrelevant. When evaluating the totality of Contagion, we show that the adversarial strategy does not affect the outcome of the execution. Here, we show that any adversarial strategy reduces to a well-defined sequence of choices. We then prove that, due to the limited knowledge of the Byzantine adversary, every choice is equivalent to a random one.

1.1.3 Security and Complexity Evaluation

In Chapters 2 to 4 we present and study the security of three algorithms: Murmur, Sieve and Contagion, building upon each other to implement probabilistic Byzantine reliable broadcast. Each algorithm is configured by one or more security parameters (sample sizes and thresholds - we discuss each in detail in the relevant chapter). Sample sizes in particular are what determine the per-process communication complexity of each algorithm: in Murmur, a correct process gossips (on average) with G other processes to disseminate the sender's message; in Sieve, a correct process samples E other processes to ensure consistency; in Contagion, a correct process decides whether or not to amplify / deliver a message based on two independent samples, respectively of size R and D.

The goal of Contagion's security analysis is to derive closed-form expressions, relating sample sizes and thresholds to the probability ϵ that Contagion will fail to uphold any of the properties of probabilistic Byzantine reliable broadcast.

We employ numerical techniques to minimize ϵ, optimizing all security parameters under the

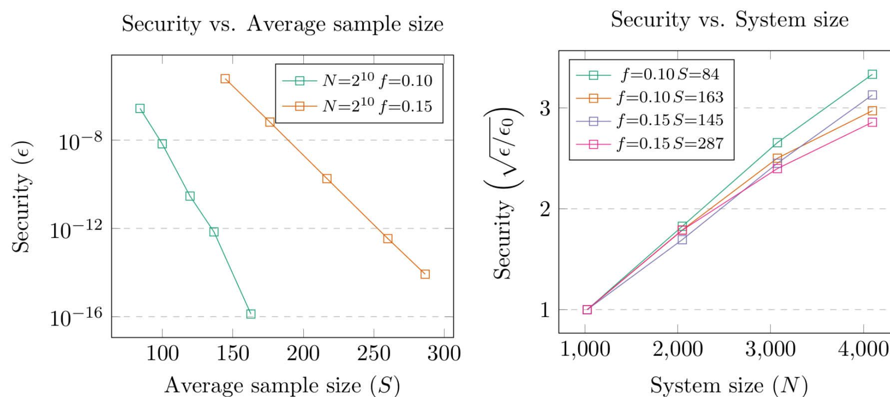


Figure 1.1: Left – ϵ-security of Contagion, as a function of the average sample size $S = \langle G, E, R, D \rangle$. We use a system size of 1024 processes and fractions of tolerated Byzantine processes $f = 0.1$ and $f = 0.15$. **Right** – Square root of the normalized ϵ-security of Contagion, as a function of the system size N, for various fractions of Byzantine processes (f) and average sample sizes (S). We normalize the values in each series by the first element of that series. All lines appearing to grow sub-linearly with a square-rooted y-axis demonstrates that the normalized ϵ security grows sub-quadratically.

constraint that the sum of all the sample sizes be constant ($G + E + R + D = const$). Because a process communicates a constant number of times with each of the processes in its samples, this corresponds to a fixed communication complexity.

For a given system size N and fraction of Byzantine processes f, we relate this per-process communication complexity to the ϵ-security of Contagion. As Figure 1.1 (left) shows, the probability ϵ of compromising the security of Contagion decays exponentially in the average sample size S. We also study how the ϵ-security of Contagion changes as a function of the system size N, for a fixed set of parameters (G, E, R, D). Figure 1.1 (right) shows that the ϵ-security is bounded by a quadratic function in N.

In summary, for a fixed security ϵ, the average sample size (and consequently, the communication complexity of our algorithm) grows logarithmically with the system size N. For a practical choice of parameters, the probability of violating the properties of our algorithm can be brought down to 10^{-16} for systems with thousands of processes.

The latency in terms of message delays between broadcasting and delivery of a message is $O(\log N / \log\log N)$. Specifically, the latency converges to $O(\log N / \log\log N)$ message delays for gossip-based dissemination with Murmur (we prove this in Section 2.4, Theorem 5), and 2 message delays in total for `Echo` (Sieve) and `Ready` (Contagion) messages.

Roadmap. We discuss related work in Section 1.2. We state our model in Section 1.3. We discuss our probabilistic broadcast implementation Murmur in Chapter 2, our probabilistic consistent broadcast implementation Sieve in Chapter 3, and our probabilistic reliable

broadcast implementation Contagion in Chapter 4. Each chapter introduces the relevant abstraction, presents the relevant pseudocode, then formally studies the algorithm's abidance to the abstraction's properties. For deterministic properties, we provide a proof of correctness. For probabilistic properties, we analitically bound the probability that the property will be violated.

1.2 Related Work

At its base, our broadcast algorithm relies on gossip. There is a great body of literature studying various aspects of gossip, proposing flavors of gossip protocols for different environments and analyzing their complexities [6, 13, 18, 19, 20, 63, 67, 73, 74, 75, 76, 77, 84, 139, 148, 157]. However, to the best of our knowledge, we propose the first highly scalable gossip-based reliable broadcast protocol resilient to Byzantine faults with a thorough probabilistic analysis.

The communication pattern in the implementation of both our Sieve and Contagion algorithms can be traced back to the Asynchronous Byzantine Agreement (ABA) primitive of Bracha and Toueg [32] and the subsequent line of work [31, 37, 110, 132]. Indeed, our echo-based mechanism in Sieve resembles algorithms from classic quorum-based systems for Byzantine consistent broadcast [131, 146]. The ready-based mechanism in Contagion is inspired by a two-phase protocol appearing in several practical (quorum-based) systems [37, 61, 111]. Compared to classic work on this topic, the key feature of Contagion and Sieve is that they replace the building block of quorum systems with stochastic samples, thus enabling better scalability for the price of abandoning deterministic guarantees.

There is significant prior work on using epidemic algorithms to implement scalable *reliable* broadcast [23, 66, 89, 106]. Under benign failures or constant churn, these algorithms ensure, with high probability, that every broadcast message reaches all or none, and that all messages from correct senders are delivered. Our goal is to additionally provide *consistency* for broadcast messages, and tolerate *Byzantine* environments [32, 111, 146]. To the best of our knowledge, we are the first to apply the epidemic sample-based methodology in this context. Our main algorithm Contagion scales well to dynamic systems of thousands of nodes, some of which may be Byzantine. This makes it a suitable choice for *permissionless* settings that are gaining popularity with the advent of blockchains [122].

Distributed clustering techniques seek to group the processes of a system into clusters, sometimes called shards or quorums, of size $O(logN)$ [14, 79, 94, 95, 134]. This line of work has various goals (e.g., leader election, "almost everywhere" agreement, building an overlay network) and they also aim for scalable solutions. The overarching principle in clustering techniques is similar to our use of samples: build each cluster in a provably random manner so that the adversary cannot dominate any single cluster. Samples in our solution are private and individual on a per-process basis, in contrast to clusters which are typically public and global for the whole system.

The idea of *communication locality* appears in the context of secure multi-party computation (MPC) protocols [29, 46, 71]. This property captures the intuition that, in order to obtain scalable distributed protocols and permit a large number of participants, it is desirable to limit the number of participants each process must communicate with. All of our three algorithms have this communication locality property, since each process coordinates only with logarithmically-sized samples. In contrast to secure MPC protocols, our algorithms have different goals, system model, or assumptions (e.g., we do not assume a client-server model [71], nor do we seek to address privacy issues). Our algorithms can be used as building blocks towards helping tackle scalability in MPC protocols, and we consider this an interesting avenue for future work.

1.3 Model

In this section we discuss our system model and assumptions, and introduce some useful notation. We assume an asynchronous message-passing system where the set Π of $N = |\Pi|$ processes partaking in an algorithm is fixed. Any two processes can communicate via a reliable authenticated point-to-point link.

We assume that each correct process has access to a local, unbiased, independent source of randomness. We assume that every correct process has direct access to an oracle Ω that, provided with an integer $n \leq N$, yields the identities of n distinct processes, chosen uniformly at random from Π. Implementing Ω is beyond the scope of this Part, but it is straightforward in practice. In a system where the set of participating processes is known, sampling reduces to picking with uniform probability an element from the set of processes. In a system without a global membership view, a (nearly) uniform sampling mechanism, *Brahms*, is available in literature due to Bortnikov *et al.* [28]. Remarkably, Brahms tolerates (slow) churn, a good fit for the usually permissionless nature of large-scale distributed systems. While we conjecture this amenability to reconfiguration would gracefully extend to our algorithms, a thorough analysis of churn is beyond the scope of this Part.

At most a fraction f of the processes are Byzantine, i.e., subject to arbitrary failures [104]. Byzantine processes may collude and coordinate their actions. Unless stated otherwise, we denote by $\Pi_C \subseteq \Pi$ the set of correct processes and by $C = |\Pi_C| = \left(1 - f\right)N$ the number of correct processes. We assume a static Byzantine adversary controlling the faulty processes, i.e., the set of processes controlled by the adversary is fixed at the beginning and does not change throughout the execution of the protocols.

We make standard cryptographic assumptions regarding the power of the adversary, namely that it cannot subvert cryptographic primitives, e.g., forge a signature. We finally assume that Byzantine processes are not aware of (1) the output of the local source of randomness of any correct process; and (2) which correct processes are communicating with each other. Both assumptions are crucial to the safety of sampling. Should the membership of a process' sample be revealed, an adversary could bias its view of the system, singling out the queried

processes with non-representative information. Anonymous communication is admittedly our system's most significant practical constraint. Even against ISP-grade adversaries, however, anonymous communication was achieved in practice (at the cost of additional latency) using techniques such as onion routing [58] or private messaging [147].

2 Murmur

In this chapter, we present the **probabilistic broadcast** abstraction and discuss its properties. We then present Murmur, an algorithm that implements probabilistic broadcast, and evaluate its **security** and **complexity** as a function of its **parameters**.

The probabilistic broadcast abstraction serves the purpose of reliably broadcasting a single message from a designated correct sender to all correct processes (**validity**, **totality**).

We use probabilistic broadcast in the implementation of Sieve (see Chapter 3) to initially distribute the message from the designated sender to all correct processes.

2.1 Interface

The **probabilistic broadcast** interface (instance pb, sender σ) exports the following **events**:

- **Request**: $\langle pb.\text{Broadcast} \mid m \rangle$: Broadcasts a message m to all processes. This is only used by σ.
- **Indication** $\langle pb.\text{Deliver} \mid m \rangle$: Delivers a message m broadcast by process σ.

For any $\epsilon \in [0, 1]$, we say that probabilistic broadcast is ϵ-secure if:

1. **No duplication**: No correct process delivers more than one message.
2. **Integrity**: If a correct process delivers a message m, and σ is correct, then m was previously broadcast by σ.
3. ϵ-**Validity**: If σ is correct, and σ broadcasts a message m, then σ eventually delivers m with probability at least $(1-\epsilon)$.
4. ϵ-**Totality**: If a correct process delivers a message, then every correct process eventually delivers a message with probability at least $(1-\epsilon)$.

2.2 Algorithm

Murmur (Algorithm 1) distributes a single message across the system by means of **gossip**: upon reception, a correct process relays the message to a set of randomly selected neighbors. The algorithm depends on one integer parameter, G (*expected gossip sample size*), whose value we discuss in Section 2.4.

Initialization. Upon initialization, (line 11) every correct process randomly samples a value $\bar{G}$ from a *Poisson* distribution with expected value G, and uses the sampling oracle Ω to select $\bar{G}$ distinct processes that it will use to initialize its **gossip sample** $\mathscr{G}$.

Link reciprocation. Once its gossip sample is initialized, a correct process sends a `GossipSubscribe` message to all the processes in $\mathscr{G}$ (line 13). Upon receiving a `GossipSubscribe` message from a process π (line 17), a correct process adds π to its own gossip sample (line 22), and sends back the gossiped message if it has already received it (line 20).

Gossip. When broadcasting the message (line 34), a correct designated sender σ signs the message and sends it to every process in its gossip sample $\mathscr{G}$ (line 28). Upon receiving a correctly signed message from σ (line 37) for the first time (this is enforced by updating the value of $delivered$, line 25), a correct process delivers it (line 30) and forwards it to every process in its gossip sample (line 28).

2.3 No duplication, integrity and validity

We start by verifying that Murmur satistifes **no duplication**, **integrity** and 0-**validity**, independently of G.

Theorem 1. *Murmur satisfies no duplication.*

Proof. Procedure $dispatch$ explicitly checks (line 25) if the variable $delivered$ is equal to $\perp$ before delivering any message. Before a message is delivered (line 30), $delivered$ is updated to a value different from $\perp$ (line 26). Therefore a correct process only delivers one message. □

Theorem 2. *Murmur satistifes integrity.*

Proof. Upon receiving a `Gossip` message, a correct process checks its signature against the public key of the designated sender σ (line 37). Moreover, if σ is correct, it only signs $message$ when broadcasting (line 34). Since we assume that cryptographic signatures cannot be forged, this implies that the message was previously broadcast by σ. □

Algorithm 1 Murmur

```
1:  Implements:
2:      ProbabilisticBroadcast, instance pb
3:
4:  Uses:
5:      AuthenticatedPointToPointLinks, instance al
6:
7:  Parameters:
8:      G: expected gossip sample size
9:
10: upon event ⟨pb.Init⟩ do
11:     𝒢 = Ω(Poisson[G]);
12:     for all π ∈ 𝒢 do
13:         trigger ⟨al.Send | π, [GossipSubscribe]⟩;
14:     end for
15:     delivered = ⊥;
16:
17: upon event ⟨al.Deliver | π, [GossipSubscribe]⟩ do
18:     if delivered ≠ ⊥ then
19:         (message, signature) = delivered;
20:         trigger ⟨al.Send | π, [Gossip, message, signature]⟩;
21:     end if
22:     𝒢 ← 𝒢 ∪ {π};
23:
24: procedure dispatch(message, signature) is
25:     if delivered = ⊥ then
26:         delivered ← (message, signature);
27:         for all π ∈ 𝒢 do
28:             trigger ⟨al.Send | π, [Gossip, message, signature]⟩;
29:         end for
30:         trigger ⟨pb.Deliver | message⟩
31:     end if
32:
33: upon event ⟨pb.Broadcast | message⟩ do                    ▷ only process σ
34:     dispatch(message, sign(message));
35:
36: upon event ⟨al.Deliver | π, [Gossip, message, signature]⟩ do
37:     if verify(σ, message, signature) then
38:         dispatch(message, signature);
39:     end if
40:
```

Theorem 3. *Murmur satisfies 0-validity.*

Proof. Upon broadcasting a message m, a correct sender calls the procedure $dispatch(m, sign(m))$ (line 34). Since $delivered$ is initialized to $\perp$, this immediately results in the delivery of m (line 30).

Since the validity property is satisfied deterministically, Murmur satisfies ϵ-validity for $\epsilon = 0$. □

2.4 Totality

We now compute, given the parameter G, the ϵ-**totality** of Murmur. To this end, we first prove some preliminary lemmas.

Lemma 1. *Let ρ and π be two correct processes, let ρ be in π's gossip sample. Then π is eventually in ρ's gossip sample.*

Proof. A gossip sample is updated only upon initialization (line 11) or when a `GossipSubscribe` message is received (line 22).

If π selected ρ upon initialization, then it also sent it a `GossipSubscribe` message (line 13). Since Byzantine network scheduling can only finitely delay the messages between correct processes, ρ eventually receives π's message (line 17) and adds π to its gossip sample.

If π received a `GossipSubscribe` message from ρ, then (line 13) ρ selected π upon initialization, which means that π is already in ρ's gossip sample. □

Definition 1 (Correct gossip network)**.** Let π, ρ be two correct processes, let $\pi \leftrightarrow \rho$ denote the condition ρ *is eventually in π's gossip sample.* Lemma 1 proves that

$$(\pi \leftrightarrow \rho) \Leftrightarrow (\rho \leftrightarrow \pi)$$

We define **correct gossip network** to be the undirected graph

$$\mathbb{G} = \left(\Pi_C, \left\{(\pi, \rho) \in \Pi_C^2 \mid \pi \leftrightarrow \rho\right\}\right) \tag{2.1}$$

Lemma 2. *If the correct gossip network is connected, then Murmur satisfies totality.*

Proof. We start by noting that a correct process eventually delivers a message (line 30) if and only if it eventually sets $delivered$ to a value different from $\perp$ (line 26).

Let π be a correct process for which eventually $delivered \neq \perp$. Upon setting $delivered \leftarrow (m \neq \perp)$, π sends m to all the processes in its gossip sample (line 28). Moreover, upon receiving a `GossipSubscribe` message *after* setting $delivered \leftarrow m$, π replies with m (line 20).

Therefore, every correct process that is eventually in π's gossip sample eventually satisfies $delivered \neq \bot$. If $\mathbb{G}$ is connected, then a path exists in $\mathbb{G}$ between π and every other correct process, and they all eventually satisfy $delivered \neq \bot$, i.e., they deliver a message. □

From Lemma 2 it follows that Murmur satisfies ϵ-totality if the probability of $\mathbb{G}$ being disconnected is at most ϵ.

Notation 1 (Binomial distribution)**.** We use $\text{Bin}[N, p]$ to denote the **binomial distribution** with N trials and p probability of success.

Notation 2 (Poisson distribution)**.** We use $\text{Poisson}[\lambda]$ to denote the **Poisson distribution** with expected value λ.

Notation 3 (Probability)**.** Let E, F be events. We use $\mathscr{P}[E]$ to denote the probability of E. We use $\mathscr{P}[E \mid F]$ to denote the probability of E, conditioned on the occurrence of F.

Let X, Y, Z be random variables. For example, we use the following expressions interchangeably:

$$\mathscr{P}[\bar{X}] \longleftrightarrow \mathscr{P}[X = \bar{X}]$$

Note how X is a random variable, while $\bar{X}$ is an element in the codomain of X. Stand-ins can be combined. For example, we use the following expressions interchangeably:

$$\begin{aligned} \mathscr{P}[\bar{X}, \bar{Y}] &\longleftrightarrow \mathscr{P}[X = \bar{X}, Y = \bar{Y}] \\ \mathscr{P}[\bar{X} \mid \bar{Y}] &\longleftrightarrow \mathscr{P}[X = \bar{X} \mid Y = \bar{Y}] \\ \mathscr{P}[\bar{X}, \bar{Y} \mid \bar{Z}] &\longleftrightarrow \mathscr{P}[X = \bar{X}, Y = \bar{Y} \mid Z = \bar{Z}] \end{aligned}$$

Stand-ins are only used to express exact values. Whenever non-trivial expressions are needed, we use their explicit form. Explicit notation and stand-ins can be combined. For example, we use the following expressions interchangeably:

$$\begin{aligned} \mathscr{P}[\bar{X} \mid Y < K] &\longleftrightarrow \mathscr{P}[X = \bar{X} \mid Y < K] \\ \mathscr{P}[\bar{X} \mid X < K] &\longleftrightarrow \mathscr{P}[X = \bar{X} \mid X < K] \end{aligned}$$

Lemma 3. *In the limit $N \to \infty$, $\mathbb{G}$ is a $G(C, p)$ Erdős–Rényi graph, with*

$$p = 1 - \left(1 - \frac{G}{N}\right)^2$$

Proof. It is a known result that, for large samples and small probabilities, a binomial distribution converges to a Poisson distribution:

$$\lim_{\substack{N\to\infty \\ Np=\text{const}}} \left[\text{Bin}[N, p](n) = \binom{N}{n} p^n (1-p)^{N-n}\right] = \left[\frac{(Np)^n}{n!} e^{-Np} = \text{Poisson}[Np](n)\right]$$

therefore, in the limit $N \to \infty$,

$$\text{Poisson}[G](n) \simeq \text{Bin}\left[N, \frac{G}{N}\right](n) \tag{2.2}$$

As we discussed in Section 2.2, a gossip sample $\mathscr{G}$ is initialized upon initialization (line 11) by first sampling a value $\bar{G}$ from a Poisson$[G]$ distribution, then selecting $\bar{G}$ distinct processes from Π with uniform probability.

Let $\pi \in \Pi_C, \rho \in \Pi$, let $\mathscr{G}_\pi^{in}$ be π's initial gossip sample, let $q = G/N$. By the law of total probability, and using Equation (2.2), we have for large N

$$\begin{aligned}
\mathscr{P}\left[\rho \in \mathscr{G}_\pi^{in}\right] &= \sum_{\bar{G}=0}^{N} \left(\mathscr{P}\left[\rho \in \mathscr{G}_\pi^{in} \mid \bar{G}\right]\mathscr{P}[\bar{G}]\right) \\
&= \sum_{\bar{G}=0}^{N} \left(\frac{\bar{G}}{N}\text{Poisson}[G](\bar{G})\right) \simeq \sum_{\bar{G}=0}^{N} \left(\frac{\bar{G}}{N}\text{Bin}[N,q](\bar{G})\right) \\
&= \sum_{\bar{G}=0}^{N} \left(\frac{\bar{G}}{N}\binom{N}{\bar{G}} q^{\bar{G}}(1-q)^{N-\bar{G}}\right) \\
&= \sum_{\bar{G}=0}^{N} \left(\frac{\bar{G}}{N}\frac{N!}{\bar{G}!(N-\bar{G})!} q^{\bar{G}}(1-q)^{N-\bar{G}}\right) \\
&= \sum_{\bar{G}=1}^{N} \left(\frac{(N-1)!}{(\bar{G}-1)!(N-\bar{G})!} q q^{\bar{G}-1}(1-q)^{N-\bar{G}}\right) \\
&= q\sum_{\bar{G}'=0}^{N-1} \left(\frac{(N-1)!}{\bar{G}'!(N-1-\bar{G}')!} q^{\bar{G}'}(1-q)^{N-1-\bar{G}'}\right) \\
&= q\sum_{\bar{G}'=0}^{N-1} \text{Bin}[N-1,q](\bar{G}') = q
\end{aligned}$$

Let $\rho_1, \ldots, \rho_R$ be distinct processes, with $R \le N$. Similar calculations yield

$$\mathscr{P}\left[\rho_1 \in \mathscr{G}_\pi^{in}, \ldots, \rho_R \in \mathscr{G}_\pi^{in}\right] = q^R \tag{2.3}$$

Equation (2.3) proves that every process $\rho \in \Pi$ has an independent probability q of being in $\mathscr{G}_\pi^{in}$. Since for any two $\pi, \xi \in \Pi_C$ we have

$$(\pi \leftrightarrow \xi) \Leftrightarrow \left(\pi \in \mathscr{G}_\xi^{in} \vee \xi \in \mathscr{G}_\pi^{in}\right)$$

we can derive the probability p of any two correct processes being connected:

$$p = 1 - (1-q)^2 = 1 - \left(1 - \frac{G}{N}\right)^2 \tag{2.4}$$

Therefore, following Equations (2.3) and (2.4), $\mathbb{G} = G(C, p)$ is an Erdős – Rényi graph with H nodes and p probability of connection between any two nodes. □

Lemma 3 allows us to bound the ϵ-totality of Murmur, given G.

Theorem 4. *Murmur satisfies ϵ_t-totality, with ϵ_t bound by*

$$\epsilon_t \leq \sum_{k=1}^{C/2} \left(\binom{C}{k} (1-p)^{k(C-k)} \right) \tag{2.5}$$

Proof. It follows immediately from Lemma 3 and a known result ([2]) on the connectivity of Erdős–Rényi graphs. □

We prove an additional result on the latency of Murmur.

Theorem 5. *The latency of Murmur is asymptotically sub-logarithmic. More formally, the diameter $D(C, G)$ of the correct gossip network limits to*

$$\lim_{C\to\infty} D(C, G) = \frac{\log(C)}{\log(2-2f) + \log(G)}$$

Proof. It is a known result ([47]) that the diameter of an Erdős–Rényi graph $G(C, p)$ converges, for $Cp \to \infty$, to $\log(C) / \log(Cp)$.

Noting that

$$\lim_{C\to\infty} 1 - \left(1 - \frac{G}{N}\right)^2 = \frac{2G}{N}$$

we get

$$\begin{aligned}
\lim_{C\to\infty} D(C, G) &= \frac{\log(C)}{\log(C) + \log(p)} \\
&= \frac{\log(C)}{\log(C) + \log(2) + \log(G) - \log(N)} \\
&= \frac{\log(C)}{\log\left(\frac{2C}{N}\right) + \log(G)} \\
&= \frac{\log(C)}{\log(2-2f) + \log(G)}
\end{aligned}$$

which proves the lemma. For a fixed security ϵ, we showed in Theorem 4 that G must scale logarithmically with the size of the system. As a result, for a fixed security ϵ, the latency scales as $O(\log(N) / \log(\log(N)))$ □

3 Sieve

In this chapter, we present the **probabilistic consistent broadcast** abstraction and discuss its properties. We then present Sieve, an algorithm that implements probabilistic consistent broadcast, and evaluate its **security** and **complexity** as a function of its **parameters**.

The probabilistic consistent broadcast abstraction allows a subset of the correct processes to agree on a single message from a potentially Byzantine designated sender. Intuitively, probabilistic consistent broadcast offers a tradeoff with respect to probabilistic broadcast. Probabilistic broadcast guarantees (**totality**) that if any correct process delivers a message, every correct process delivers a message. Probabilistic consistent broadcast, instead, guarantees (**consistency**) that, even if the sender is Byzantine, no two correct processes deliver different messages. However, if the sender is Byzantine, it may happen with a non-negligible probability that only an intermediate fraction of the correct processes deliver the message.

We use probabilistic consistent broadcast in the implementation of Contagion (see Chapter 4) as a way to consistently broadcast messages.

3.1 Interface

The **probabilistic consistent broadcast** interface (instance pcb, sender σ) exposes the following two **events**:

- **Request**: $\langle pcb.\text{Broadcast} \mid m\rangle$: Broadcasts a message m to all processes. This is only used by σ.
- **Indication**: $\langle pcb.\text{Deliver} \mid m\rangle$: Delivers a message m broadcast by process σ.

For any $\epsilon \in [0,1]$, we say that probabilistic consistent broadcast is ϵ-secure if:

1. **No duplication**: No correct process delivers more than one message.

2. **Integrity**: If a correct process delivers a message m, and σ is correct, then m was previously broadcast by σ.

3. ϵ-**Total validity**: If σ is correct, and σ broadcasts a message m, every correct process eventually delivers m with probability at least $(1-\epsilon)$.

4. ϵ-**Consistency**: Every correct process that delivers a message delivers the same message with probability at least $(1-\epsilon)$.

3.2 Algorithm

Algorithm 2 Procedure $sample$**.**

1: **procedure** *sample*($message, size$) **is**
2: $\psi = \emptyset$;
3: **for** $size$ **times do**
4: $\psi \leftarrow \psi \cup \Omega(1)$;
5: **end for**
6: **for all** $\pi \in \psi$ **do**
7: **trigger** $\langle al.\text{Send} \mid \pi, [message] \rangle$;
8: **end for**
9: **return** ψ;
10:

Algorithm 2 implements a $sample$ procedure that we use both in the implementation of Sieve and Contagion. Procedure $sample(message, size)$ uses Ω to pick $size$ processes with replacement, and sends them $message$.

Algorithm 3 implements Sieve. Sieve consistently distributes a single message across the system as follows:

- Initially, probabilistic broadcast distributes potentially conflicting copies of the message to every correct process.
- Upon receiving a message m from probabilistic broadcast, a correct process issues an Echo message for m.
- Upon receiving enough Echo messages for the message m it Echoed, a correct process delivers m.

A correct process collects Echo messages from a randomly selected *echo sample* of size E, and delivers the message it Echoed upon receiving $\hat{E}$ Echoes for it. We discuss the values of the two parameters of Sieve in Section 3.10.

Algorithm 3 Sieve

1: **Implements:**
2: ProbabilisticConsistentBroadcast, **instance** pcb
3:
4: **Uses:**
5: AuthenticatedPointToPointLinks, **instance** al
6: ProbabilisticBroadcast, **instance** pb
7:
8: **Parameters:**
9: E: echo sample size
10: $\hat{E}$: delivery threshold
11:
12: **upon event** $\langle pcb.\text{Init}\rangle$ **do**
13: $echo = \bot$; $delivered = \texttt{False}$; $\tilde{\mathscr{E}} = \emptyset$;
14:
15: $\mathscr{E} = sample(\texttt{EchoSubscribe}, E)$;
16: $replies = \{\bot\}^E$;
17:
18: **upon event** $\langle al.\text{Deliver} \mid \pi, [\texttt{EchoSubscribe}]\rangle$ **do**
19: **if** $echo \neq \bot$ **then**
20: $(message, signature) = echo$;
21: **trigger** $\langle al.\text{Send} \mid \pi, [\texttt{Echo}, message, signature]\rangle$;
22: **end if**
23: $\tilde{\mathscr{E}} \leftarrow \tilde{\mathscr{E}} \cup \{\pi\}$;
24:
25: **upon event** $\langle pcb.\text{Broadcast} \mid message\rangle$ **do** ▷ only process σ
26: **trigger** $\langle pb.\text{Broadcast} \mid [\texttt{Send}, message, sign(message)]\rangle$;
27:
28: **upon event** $\langle pb.\text{Deliver} \mid [\texttt{Send}, message, signature]\rangle$ **do**
29: **if** $verify(\sigma, message, signature)$ **then**
30: $echo \leftarrow (message, signature)$;
31: **for all** $\rho \in \tilde{\mathscr{E}}$ **do**
32: **trigger** $\langle al.\text{Send} \mid \rho, [\texttt{Echo}, message, signature]\rangle$;
33: **end for**
34: **end if**
35:
36: **upon event** $\langle al.\text{Deliver} \mid \pi, [\texttt{Echo}, message, signature]\rangle$ **do**
37: **if** $\pi \in \mathscr{E}$ **and** $replies[\pi] = \bot$ **and** $verify(\sigma, message, signature)$ **then**
38: $replies[\pi] \leftarrow (message, signature)$;
39: **end if**
40:
41: **upon** $\left|\{\rho \in \mathscr{E} \mid replies[\rho] = echo\}\right| \geq \hat{E}$ **and** $delivered = \texttt{False}$ **do**
42: $delivered \leftarrow \texttt{True}$;
43: **trigger** $\langle pcb.\text{Deliver} \mid message\rangle$;
44:

Sampling. Upon initialization (line 12), a correct process randomly selects an **echo sample** $\mathscr{E}$ of size E. Samples are selected with replacement by repeatedly calling Ω (Algorithm 2, line 4). A correct process sends an `EchoSubscribe` message to all the processes in its echo sample (Algorithm 2, line 7).

Publish-subscribe. Unlike in the deterministic version of `Authenticated Echo Broadcast`, where a correct process broadcasts its `Echo` messages to the whole system, here each process only listens for messages coming from its echo sample (line 37).

A correct process maintains an **echo subscription set** $\tilde{\mathscr{E}}$. Upon receiving an `EchoSubscribe` message from a process π, a correct process adds π to $\tilde{\mathscr{E}}$ (line 23). If a correct process receives an `EchoSubscribe` message *after* publishing its `Echo` message, it also sends back the previously published message (line 21).

A correct process only sends its `Echo` messages (line 32) to its echo subscription set.

Echo. The designated sender σ initially broadcasts its message using probabilistic broadcast (line 26). Upon pb.Delivery of a message m (correctly signed by σ) (line 28), a correct process sends an `Echo` message for m to all the nodes in its echo subscription set (line 32).

Delivery. A correct process π that `Echo`ed a message m delivers m (line 43) upon collecting at least $\hat{E}$ `Echo` messages for m (line 41) from the processes in its echo sample.

3.3 No duplication and integrity

We start by verifying that Sieve satisfies both **no duplication** and **integrity**.

Theorem 6. *Sieve satisfies no duplication.*

Proof. A message is delivered (line 43) only if the variable $delivered$ is equal to `False` (line 41). Before any message is delivered, $delivered$ is set to `True`. Therefore no more than one message is ever delivered. □

Theorem 7. *Sieve satisfies integrity.*

Proof. Upon receiving an `Echo` message, a correct process checks its signature against the public of the designated sender σ (line 37), and the $(message, signature)$ pair is added to the $replies$ variable only if this check succeeds. Moreover, a message is delivered only if it is represented at least $\hat{E} > 0$ times in $replies$ (line 41).

If σ is correct, it only signs $message$ when broadcasting (line 26). Since we assume that cryptographic signatures cannot be forged, this implies that the message was previously

broadcast by σ. □

3.4 Total validity

We now compute, given E and $\hat{E}$, the ϵ-**total validity** of Sieve. To this end, we prove some preliminary lemmas.

Lemma 4. *In an execution of Sieve, if pb does not satisfy totality, then pcb does not satisfy total validity.*

Proof. A correct process delivers a message (line 43) only if the $echo$ variable is different from $\perp$. Moreover, the $echo$ variable is set to a value different from $\perp$ (line 30) only upon pb.Delivery of a message (line 28).

Let m be the message broadcast by the correct sender σ. If pb does not satisfy totality, then at least one correct process never sets $echo$ to m. Therefore, at least one correct process does not deliver the m, and the total validity of pcb is comrpomised. □

Lemma 5. *In an execution of Sieve, if pb satisfies totality and no correct process has more than $E - \hat{E}$ Byzantine processes in its echo sample, then pcb satisfies total validity.*

Proof. Let m be the message broadcast by the correct sender σ. Since pb satisfies totality (it always satisfies validity), every correct process eventually issues an $\texttt{Echo}(m)$ message (i.e., an Echo message for m) (line 32).

Let π be a correct process that has no more than $E - \hat{E}$ Byzantine processes in its echo sample. Obviously, π has at least $\hat{E}$ correct processes in its echo sample. Therefore, π eventually receives at least $\hat{E}$ $\texttt{Echo}(m)$ messages (line 36), and delivers m (line 41). □

Lemmas 4 and 5 allow us to bound the ϵ-total validity of Sieve, given E and $\hat{E}$.

Theorem 8. *Sieve satisfies ϵ_v-total validity, with*

$$\begin{aligned} \epsilon_v &\leq \epsilon_t^{pb} + \left(1 - \epsilon_t^{pb}\right)\left(1 - (1 - \epsilon_o)^C\right) \\ \epsilon_o &= \sum_{\bar{F}=E-\hat{E}+1}^{E} Bin[E, f](\bar{F}) \end{aligned} \tag{3.1}$$

if the underlying abstraction of probabilistic broadcast satisfies ϵ_t^{pb}-totality.

Proof. Following from Lemmas 4 and 5, the total validity of pcb can be compromised only if the totality of pb is compromised as well, or if at least one correct process has more than $E - \hat{E}$ Byzantine processes in its echo sample.

Since procedure *sample* independently picks E processes with replacement, each element of a correct process' echo sample has an independent probability f of being Byzantine, i.e., the number of Byzantine processes in a correct echo sample is binomially distributed.

Therefore, a correct process has a probability ϵ_o of having more than $E-\hat{E}$ Byzantine processes in its echo sample. Since every correct process picks its echo sample independently, the probability of at least one correct process having more than $E-\hat{E}$ Byzantine processes in its echo sample is $1-(1-\epsilon_o)^C$. □

3.5 Preliminary lemmas

In order to compute an upper bound for the probability of the consistency of Sieve being compromised, we will make use of some preliminary lemmas. The statements of these lemmas are independent from the context of Sieve. For the sake of readability, we therefore gather them in this section, and use them throughout the rest of this chapter.

Lemma 6. *Let $A, B \in \mathbb{N}$, let $x, y \in \mathbb{N}$ such that $x + y \leq B$. Let X, Y be random variables defined by*

$$\begin{aligned} \mathscr{P}\left[\bar{X}\right] &= Bin\left[A, \frac{x}{B}\right](\bar{X}) \\ \mathscr{P}\left[\bar{Y} \mid \bar{X}\right] &= Bin\left[A - \bar{X}, \frac{y}{B-x}\right](\bar{Y}) \end{aligned}$$

We have

$$\mathscr{P}[X + Y = K] = Bin\left[A, \frac{x+y}{B}\right](K)$$

Proof. Since X is binomially distributed, it can be expressed as a sum of independent Bernoulli random variables:

$$\begin{aligned} X &= X_1 + \ldots + X_A \\ X_i &\sim \text{Bern}\left[\frac{x}{B}\right] \end{aligned}$$

Given the value of $\bar{X}$, Y is also binomially distributed with probability $y/(B-x)$ and $E-\bar{X}$ trials. We can therefore express Y as the sum of E Bernoulli variables $Y_1, \ldots, Y_E$:

$$\begin{aligned} Y &= Y_1 + \ldots + Y_E \\ \mathscr{P}\left[Y_i = 1 \mid \bar{X}_i\right] &= \begin{cases} 0 & \text{iff } \bar{X}_i = 1 \\ \frac{y}{B-x} & \text{otherwise} \end{cases} \end{aligned}$$

We indeed note how, out of $Y_1, \ldots, Y_E$:

- Only $E-\bar{X}$ variables have a non-null probability of being equal to 1.

- Those variables that have a non-null probability of being equal to 1 have a probability $y/(B-x)$ of being equal to 1.

We therefore have

$$X+Y=(X_1+Y_1)+\ldots+(X_A+Y_A)$$

and from the law of total probability we have

$$\begin{aligned}
\mathscr{P}[X_i+Y_i=1] &= \mathscr{P}[X_i=1]+\mathscr{P}[Y_i=1\mid X_i=0]\mathscr{P}[X_i=0]\\
&= \frac{x}{B}+\left(1-\frac{x}{B}\right)\left(\frac{y}{B-x}\right)\\
&= \frac{x}{B}+\left(\frac{(B-x)}{B}\frac{y}{(B-x)}\right)\\
&= \frac{x+y}{B}
\end{aligned}$$

therefore

$$(X_i+Y_i)\sim \mathrm{Bern}\left[\frac{x+y}{B}\right]$$

which proves the lemma. □

Lemma 7. *Let $A,B\in\mathbb{N}$ such that $A\geq B$, let $p\in[0,1]$. Let $X_1,\ldots,X_B$ be random variables defined by*

$$X_i\sim Bin\left[A-i,p\right]$$

We have that

$$\mathscr{P}[X_i\geq B-i]$$

is an increasing function of i.

Proof. We prove the lemma by induction by showing that, for any $i<B$,

$$\mathscr{P}[X_i\geq B-i]\leq\mathscr{P}[X_{i+1}\geq B-(i+1)]$$

In order to obtain the above, we expand

$$\begin{aligned}
&\mathscr{P}[X_i\geq B-i]-\mathscr{P}[X_{i+1}\geq B-i-1]\\
=&\sum_{n=B-i}^{A-i}\left(\frac{(A-i)!}{(A-i-n)!n!}p^n(1-p)^{A-i-n}\right)\\
&-\sum_{n=B-i-1}^{A-i-1}\left(\frac{(A-i-1)!}{(A-i-1-n)!n!}p^n(1-p)^{A-i-1-n}\right)\\
=&(\star_1)
\end{aligned}$$

By shifting the index in the second sum we get

$$\begin{aligned}
(\star_1) &= \sum_{n=B-i}^{A-i}\left(\frac{(A-i)!}{(A-i-n)!n!}p^n(1-p)^{A-i-n}\right)\\
&\quad - \sum_{n=B-i}^{A-i}\left(\frac{(A-i-1)!}{(A-i-1-(n-1))!(n-1)!}\right.\\
&\qquad\qquad \left. p^{(n-1)}(1-p)^{A-i-1-(n-1)}\right)\\
&= \sum_{n=B-i}^{A-i}\left(\frac{(A-i)!}{(A-i-n)!n!}p^n(1-p)^{A-i-n}\right.\\
&\qquad\qquad \left. -\frac{(A-i)!n}{(A-i)(A-i-n)!n!}\frac{p^n}{p}(1-p)^{A-i-n}\right)\\
&= \sum_{n=B-i}^{A-i}\left(\left(\frac{(A-i)!}{(A-i-n)!n!}p^n(1-p)^{A-i-n}\right)\left(1-\frac{n}{(A-i)p}\right)\right)\\
&= (\star_2)
\end{aligned}$$

and by letting $N = A-i$, $M = B-i$ we get

$$(\star_2) = \sum_{n=M}^{N}\left(\mathrm{Bin}[N,p](n)\left(1-\frac{n}{Np}\right)\right) = (\star_3)$$

Noticing that $(1 - n/Np)$ is positive for $n < Np$, we have

$$\begin{aligned}
(\star_3) &\le \sum_{n=0}^{N}\left(\mathrm{Bin}[N,p](n)\right) - \frac{1}{Np}\sum_{n=0}^{N}\left(n\mathrm{Bin}[N,p](n)\right)\\
&= 1-\frac{Np}{Np} = 0
\end{aligned}$$

which proves the lemma. □

Notation 4 (Ranges)**.** Let $a, b \in \mathbb{N}$, with $b \ge a$. We use $a..b$ to denote the range of integers $\{a, \ldots, b\}$.

Lemma 8. *Let $f, g : 0..K \to \mathbb{R}$, with f increasing, g positive and*

$$\sum_{x=0}^{K} g(x) = 1$$

we have

$$\sum_{x=0}^{K}\left(f(x)g(x)\right) \ge \sum_{x=0}^{K-1}\left(\frac{f(x)g(x)}{1-g(K)}\right)$$

Proof. We have

$$
\begin{aligned}
& \sum_{x=0}^{K} \big(f(x)g(x)\big) - \sum_{x=0}^{K-1} \left(\frac{f(x)g(x)}{1-g(K)}\right) \\
= \quad & f(K)g(K) + \sum_{x=0}^{K-1} \big(f(x)g(x)\big)\left(1 - \frac{1}{1-g(K)}\right) \\
= \quad & f(K)g(K) - \sum_{x=0}^{K-1} \big(f(x)g(x)\big)\left(\frac{1-\big(1-g(K)\big)}{1-g(K)}\right) \\
= \quad & g(K)\left(f(K) - \frac{1}{1-g(K)} \sum_{x=0}^{K-1} \big(f(x)g(x)\big)\right) \\
= \quad & \frac{g(K)}{1-g(K)}\left(f(K) - f(K)g(K) - \sum_{x=0}^{K-1} \big(f(x)g(x)\big)\right) \\
= \quad & \frac{g(K)}{1-g(K)}\left(f(K) \sum_{x=0}^{K} \big(f(x)g(x)\big)\right) \\
= \quad & (\star_1)
\end{aligned}
$$

and noting that $g(K) \geq 0$, $1-g(K) \geq 0$, and f is increasing, we have

$$
(\star_1) \geq \frac{g(K)}{1-g(K)}\left(f(K) - f(K)\sum_{x=0}^{K} g(x)\right) = (\star_2)
$$

and since $\sum g(x) = 1$ we get

$$
\frac{g(K)}{1-g(K)}\big(f(k) - f(K)\big) = 0
$$

□

Corollary 1. *Let $f, g : 0..K \to \mathbb{R}$, with f increasing, g positive and*

$$
\sum_{x=0}^{K} g(x) = 1
$$

for any $l \in 0..(K-1)$, we have

$$
\sum_{x=0}^{K} \big(f(x)g(x)\big) \geq \frac{\sum_{x=0}^{K-l} f(x)g(x)}{\sum_{x=0}^{K-l} g(x)}
$$

Proof. It follows immediately from applying Lemma 8 l times. □

Lemma 9. *Let $f : -1..C \to \mathbb{R}$, let $g, h : -1..C \to [0,1]$, with:*

- *f decreasing.*
- *g, h increasing.*
- *$g(x) \leq h(x)$ for all x.*

- $g(-1) = h(-1) = 0$.
- $g(C) = h(C) = 1$.

We have

$$\sum_{x=0}^{C} \big(f(x)\big(g(x) - g(x-1)\big)\big) \leq \sum_{x=0}^{C} \big(f(x)(h(x) - h(x-1))\big)$$

Proof. We have

$$\begin{aligned}
&\sum_{x=0}^{C} f(x)\big(g(x) - g(x-1)\big) - \sum_{x=0}^{C} f(x)(h(x) - h(x-1)) \\
= &\sum_{x=0}^{C} f(x)\big(\big(g(x) - h(x)\big) - \big(g(x-1) - h(x-1)\big)\big) \\
= &\sum_{x=0}^{C} f(x)\big(g(x) - h(x)\big) - \sum_{x=0}^{C} f(x)\big(g(x-1) - h(x-1)\big) \\
= &(\star_1)
\end{aligned}$$

By shifting the index in then second sum we get

$$\begin{aligned}
(\star_1) \quad = &\sum_{x=0}^{C} f(x)\big(g(x) - h(x)\big) - \sum_{x=-1}^{C-1} f(x+1)\big(g(x) - h(x)\big) \\
= &\sum_{x=0}^{C-1} \big(f(x) - f(x+1)\big)\big(g(x) - h(x)\big) \\
&+ f(C)\big(g(C) - h(C)\big) - f(-1)\big(g(-1) - h(-1)\big) \\
= &(\star_2)
\end{aligned}$$

and by noting that:

- Since f is decreasing, $f(x) - f(x+1) \geq 0$.
- By hypothesis, $g(x) - h(x) \leq 0$.
- By hypothesis, $g(C) - h(C) = 1 - 1 = 0$.
- By hypothesis, $g(-1) - h(-1) = 0 - 0 = 0$.

Consequently, all the terms of the sum in $(\star_2)$ are negative, and the two terms out of the sum are null. Therefore, $(\star_2) \leq 0$. □

Lemma 10. *Let $N \in \mathbb{N}$, let $K < N$, let $h, p_1, \ldots, p_T \in [0,1]$ such that*

$$h = \sum_i p_i \leq \frac{K - \sqrt{K}}{N}$$

let $X_1, \ldots, X_T$ be independent random variables defined by

$$\mathscr{P}[\bar{X}_i] = Bin[N, p_i](\bar{X}_i)$$

We have

$$\mathscr{P}\left[\bigvee_i (X_i > K)\right] \leq \left(\frac{eNh}{K}\right)^K e^{-Nh}$$

Proof. Let $p \in [0,1]$, let $X \sim \text{Bin}[N, p]$. From the multiplicative form of the Chernoff bound we have

$$\begin{aligned}
\mathscr{P}[X > K] &= \mathscr{P}[X > (1+\delta)\mu] < \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^\mu \\
\delta &= \left(\frac{K}{Np} - 1\right) \\
\mu &= Np
\end{aligned}$$

From the above follows

$$\begin{aligned}
\mathscr{P}[X > K] &< \left(\frac{\exp\left(\frac{K}{Np} - 1\right)}{\left(\frac{K}{Np}\right)^{\frac{K}{Np}}}\right)^\mu \\
&= \exp\left(Np\left(\frac{K}{Np} - 1 - \frac{K}{Np}\log\left(\frac{K}{Np}\right)\right)\right) \\
&= \exp\left(K - Np - K\log\left(\frac{K}{Np}\right)\right) \\
&= \underbrace{\exp\left(K - K\log K + K\log N\right)}_{(\star_a)}\underbrace{\exp\left(K\log p - Np\right)}_{(\star_b)}
\end{aligned}$$

We now study the domain where $(\star_b)$ is convex:

$$\frac{\partial^2}{\partial^2 p}\exp\left(K\log p - Np\right) = \underbrace{p^{K-2}e^{-Np}}_{\geq 0}\left(K^2 - K(2Np+1) + N^2p^2\right)$$

Therefore we require

$$N^2p^2 - (2KN)p + \left(K^2 - K\right) \geq 0$$

Which reduces to

$$\begin{aligned} p &\leq \frac{2KN - \sqrt{4K^2N^2 - 4\left(N^2K^2 - N^2K\right)}}{2N^2} \\ &= \frac{K - \sqrt{K}}{N} \end{aligned}$$

From Boole's inequality we have

$$\mathscr{P}\left[\bigvee_i (X_i > K)\right] \leq \sum_i \mathscr{P}[X_i > K] = (\star_1)$$

which we can expand into

$$(\star_1) = \underbrace{\exp\left(K - K\log K + K\log N\right)}_{(\star_a)} \sum_i \underbrace{\exp\left(K\log p_i - Np_i\right)}_{(\star_b)} = (\star_2)$$

As we established, $(\star_b)$ is convex on the range $\left[0, \sum_i p_i\right]$. Consequently,

$$\begin{aligned} (\star_2) &\leq \exp\left(K - K\log K + K\log N\right)\exp\left(K\log h - Nh\right) \\ &= \left(\frac{eNh}{K}\right)^K e^{-Nh} \end{aligned}$$

which proves the lemma. □

3.6 Simplified Sieve

In this section, we introduce Simplified Sieve, a modified version of Sieve.

Simplified Sieve is a strawman both from a performance and a safety point of view. Indeed, on the one hand Simplified Sieve has $O(N^2)$ per-process communication complexity, which makes it unfit for any real-world, scalable deployment. On the other, we prove that it is strictly easier for any Byzantine adversary to compromise the consistency of Simplified Sieve than that of Sieve.

Unlike Sieve, however, Simplified Sieve allows for an analytic probabilistic analysis. A critical goal of this chapter is to compute a bound ϵ_c on the probability of compromising the consistency of Simplified Sieve. Since the consistency of Simplified Sieve is weaker than that of Sieve, ϵ_c is also a bound on the probability of compromising the consistency of Sieve.

3.6.1 Consistency-only broadcast

Simplified Sieve implements consistency-only broadcast, a minimal version of the probabilistic consistent broadcast abstraction, designed to only provide ϵ-consistency. In particular, we

drop the no duplication property, i.e., we allow a correct process to deliver more than one message.

The **consistency-only broadcast** interface (instance cob, sender σ) exposes the following two **events**:

- **Request**: $\langle cob.\text{Broadcast} \mid m \rangle$: Broadcasts a message m to all processes. This is only used by σ.
- **Indication** $\langle cob.\text{Deliver} \mid m \rangle$: Delivers a message m broadcast by process σ.

For any $\epsilon \in [0,1]$, we say that consistency-only broadcast is ϵ-secure if:

1. ϵ-**Consistency**: With probability at least $(1-\epsilon)$, at most one message m exists, such that m is delivered by any correct process.

We note how the above definition of ϵ-consistency is equivalent to the one we provided in Section 3.1, but adapted for a context where no duplication is not guaranteed. In consistency-only broadcast, consistency is compromised even if a single correct process delivers two or more different messages.

3.6.2 Byzantine oracle

In order to implement Simplified Sieve, we make an additional assumption about the system:

- (**Byzantine oracle**) Every correct process has direct access to an oracle Ψ that, provided with a process π, returns `True` if π is Byzantine, and `False` if π is correct.

This assumption is obviously unsatisfiable in any realistic distributed system. Indeed, a system where every faulty process is publicly flagged can hardly be considered being subject to arbitrary failures. It is therefore critical to underline that Assumption 3.6.2 is **not** a requirement for the implementation of Sieve. Indeed, no correct process invokes Ψ throughout any execution of Algorithm 3. Assumption 3.6.2 is purely a theoretical artifice to aid in our proof of correctness.

3.6.3 Algorithm

Before introducing the design principles behind Simplified Sieve, we prove a simple preliminary result.

Lemma 11. *No execution of probabilistic broadcast results in more than C different messages being delivered.*

Proof. Following from Theorem 1, probabilistic broadcast satisfies no duplication, i.e., no correct process delivers more than one message. As we discussed in Section 1.3, the system is composed of C correct processes. □

Since the set of messages that are pb.Delivered by at least one correct process has no more than C elements, and noting that a correct process pcb.Delivers a message m only if it pb.Delivered m, it is not restrictive to introduce the following definition.

Definition 2 (Message)**.** A **message** is an element of the set

$$\mathcal{M} = 1..C$$

Algorithm 4 Procedure $mimic$**.**

```
1:  procedure correct() is
2:      do
3:          ρ = Ω(1)
4:      until Ψ(ρ) = False
5:      return ρ;
6:
7:  procedure mimic(reference) is
8:      ψ = ∅;
9:      for all ρ ∈ reference do
10:         if Ψ(ρ) = True then
11:             ψ ← ψ ∪ {ρ};
12:         else
13:             ψ ← ψ ∪ correct();
14:         end if
15:     end for
16:     return ψ;
17:
```

Algorithm 5 implements Simplified Sieve. Simplified Sieve bears multiple differences to Sieve:

- A correct process can deliver more than one message. No correct process, however, delivers the same message more than once.
- In order to cob.Deliver a message, a correct process does not need to pb.Deliver any message.
- A correct process maintains C **echo samples** $\mathscr{E}[1..C]$. The Echo messages collected from the processes in the i-th echo sample $\mathscr{E}[i]$ determine whether or not message $i \in \mathcal{M}$ is delivered.
- Echo messages have two fields: $sample$ and $message$. Intuitively, an Echo(s, m) message (i.e., an Echo message with fields s and m) represents the following statement: "within the context of message s, consider my Echo to be for message m".

Algorithm 5 Simplified Sieve

```
1:  Implements:
2:      ConsistencyOnlyBroadcast, instance cob
3:
4:  Uses:
5:      AuthenticatedPointToPointLinks, instance al
6:      ProbabilisticBroadcast, instance pb
7:
8:  Parameters:
9:      E: echo sample size
10:     Ê: delivery threshold
11:
12: upon event ⟨cob.Init⟩ do
13:     delivered = {False}^C; reveal = {False}^C;
14:     revealed = {False}^C;
15:     replies = {⊥}^(C×E);                          ▷ C × E table filled with ⊥.
16:     𝓔 = {∅}^C;
17:     𝓔[1] ← sample(EchoSubscribe, E);
18:
19:     for j ∈ 2..C do
20:         𝓔[j] ← mimic(𝓔[1]);
21:     end for
22:
23: upon event ⟨cob.Broadcast | message⟩ do                 ▷ only process σ
24:     trigger ⟨pb.Broadcast | [Send, message]⟩;
25:
26: upon event ⟨pb.Deliver | [Send, message]⟩ do
27:     for all ρ ∈ Π do
28:         for all sample ∈ 𝓜 do
29:             trigger ⟨al.Send | ρ, [Echo, sample, message]⟩;
30:         end for
31:     end for
32:
33: upon event ⟨al.Deliver | ρ, [Echo, sample, message]⟩ do
34:     if ρ ∈ 𝓔[sample] and replies[sample][ρ] = ⊥ then
35:         replies[sample][ρ] ← message;
36:         revealed[sample] ← False;
37:     end if
38:
39: upon exists message such that |{ρ ∈ 𝓔[message] | replies[message][ρ] = message}| ≥
    Ê and delivered[message] = False do
40:     delivered[message] ← True;
41:     reveal[message] ← True;
42:     trigger ⟨cob.Deliver | message⟩;
43:
```

```
44: upon exists message such that reveal[message] = True and revealed[message] =
    False do
45:     revealed[message] ← True;
46:     sample = {ρ ∈ ℰ[message] | replies[message][ρ] ≠ ⊥};
47:     for all π ∈ Π do
48:         trigger ⟨al.Send | π, [Reveal, message, sample]⟩;
49:     end for
50:
51: upon event ⟨al.Deliver | ρ, [Reveal, message, sample]⟩ do
52:     if Ψ(ρ) = False then
53:         reveal[message] ← True;
54:     end if
55:
```

Upon pb.Delivering a message m, a correct process sends C `Echo` messages to each other process, one `Echo`(s, m) message for every $s \in \mathcal{M}$. In other words, the correct behavior is to echo m across all contexts $s \in \mathcal{M}$. A Byzantine process, however, can in principle send to the same process a set of `Echo` messages echoing different messages in different contexts (e.g, `Echo`(s, m) and `Echo`$(s', m' \neq m)$).

- When a correct process π collects at least $\hat{E}$ `Echo`(m, m) messages from the processes in $\mathcal{E}[m]$, π delivers m.

Mimic. Algorithm 4 presents two utility procedures for manipulating samples with respect to their Byzantine component:

- ($correct$, line 1) Procedure $correct$ returns a correct process, picked with uniform probability. It does so by invoking Ω to select a process ρ with uniform probability (line 3), then using Ψ to pick again if ρ is Byzantine (line 4).
- ($mimic$, line 7) Provided with a sample $reference$, procedure $mimic$ returns a sample ψ that shares with $reference$ all Byzantine processes. It does so by looping over each process ρ in $reference$. If ρ is Byzantine (line 11), ρ is added to ψ. If ρ is correct (line 13), procedure $correct()$ is used to add a random correct process to ψ.

Samples. Upon initialization (line 12), a correct process initializes C echo samples $\mathcal{E}[1..C]$ that share the same set of Byzantine processes. It does so by using procedure $sample(\ldots)$ to randomly pick $\mathcal{E}[1]$ (line 17), then using $mimic(\mathcal{E}[1])$ to pick samples 2 to C (line 20).

We underline how $\mathcal{E}[1]$ is selected using the $sample$ procedure we defined in Algorithm 2. As a result, upon initialization, a correct process sends an `EchoSubscribe` message to each process in $\mathcal{E}[1]$. However, a correct process does not handle the al.Delivery of an `EchoSubscribe` message. This is done on purpose. The only goal of those `EchoSubscribe` messages is to let

the Byzantine adversary know which Byzantine processes are in $\mathcal{E}[1]$ (and, consequently, in every other sample).

Broadcast. Upon cob.Broadcasting a message $message$ (line 23), the correct designated sender uses pb.Broadcast to distribute $message$.

Echo. When a correct process pb.Delivers a message $message$ (line 26), it sends to each process ρ an Echo($sample, message$) message, for every $sample$ in $\mathcal{M}$ (line 29). In other words, the correct behavior of a correct process that pb.Delivered $message$ is to echo $message$ across all samples.

We note how Simplified Sieve does not make use of echo subscription sets. A correct process sends its Echo messages to every process in the system. The goal of Simplified Sieve, indeed, is not performance, but probabilistic tractability.

Delivery. A correct process maintains a table $replies$ to keep track of the Echo messages received by each node in its echo samples. Upon receiving an Echo($sample$, $message$) message from a process ρ for the first time (line 33), if ρ is in $\mathcal{E}[sample]$, a correct process sets $replies[sample][\rho]$ to $message$ (line 35).

Upon receiving at least $\hat{E}$ Echo($message$, $message$) messages from the processes in $\mathcal{E}[message]$ (line 39) (this is checked using the $replies$ table), a correct process cob.Delivers $message$ (line 42).

Reveal. A correct process maintains a $reveal$ array to keep track of which echo samples it should *reveal*. When, for some $message$, $reveal[message]$ = True (line 44), a correct process sends to every process a Reveal message, containing the set of processes in $\mathcal{E}[message]$ that issued an Echo($message$, $message'$) message for some $message' \in \mathcal{M}$ (line 48). In other words, whenever $reveal[message]$ = True, a correct process reveals the set of processes in its echo sample for $message$ that issued a an Echo message for that sample.

If, after revealing its sample for $message$, a correct process receives additional Echo messages from the processes in $\mathcal{E}[message]$, the reveal procedure is performed again. This is enforced by setting a $revealed$ flag back to False (line 36) every time a new Echo message is received.

A correct process sets $reveal[message]$ to True under two circumstances: when it cob.Delivers $message$ (line 41) and when it receives a Reveal message for $message$ from a correct process (line 53). As a result, whenever any correct process delivers $message$, every correct process reveals its sample for $message$, regardless of whether or not it delivered $message$.

Like EchoSubscribe, the Reveal message serves the only purpose to provide information to

the Byzantine adversary.

3.7 Adversarial execution

In this section, we define the model underlying an adversarial execution of Sieve and Simplified Sieve, and identify the set of Byzantine adversaries for either algorithm. Here, a Byzantine adversary is an agent that acts upon a system with the goal of compromising its consistency. Throughout the rest of this chapter, we use the term **pcb adversary** to denote a Byzantine adversary for Sieve, and the term **cob adversary** (or just **adversary**) to denote a Byzantine adversary for Simplified Sieve.

The main goal of this section is to formalize the information available both to the pcb and the cob adversary, and the set of actions that they can perform on the system throughout an adversarial execution of either algorithm.

Throughout the rest of this chapter, we bound the probability of compromising the consistency of Sieve by assuming that, if the totality of pb is compromised, then the consistency of pcb is compromised as well. In what follows, therefore, we assume that pb satisfies totality.

3.7.1 Model (Sieve)

Let π be any correct process. We make the following assumptions about an adversarial execution of Sieve:

- As we established in Section 1.3, the pcb adversary does not know which correct processes are in π's echo sample. The pcb adversary knows, however, which Byzantine processes are in π's echo sample.
- At any time, the pcb adversary knows if π delivered a message. If π delivered a message, then the pcb adversary knows which message did π deliver.
- The pcb adversary can cause π to pb.Deliver any message. As we established with Theorem 1, π will, however, pb.Deliver only one message throughout an execution of Sieve.

Throughout an adversarial execution of Sieve, an adversary performs a sequence of minimal operations on the system. Each operation consists of either of the following:

- Selecting a correct process that did not pb.Deliver any message, and causing it to pb.Deliver a message.
- Selecting a Byzantine process and causing it to send an `Echo` message to a correct process.

As a result of each operation, zero or more correct processes deliver a message. The pcb adversary is successful if, at the end of the adversarial execution, at least two different messages are delivered by at least one correct process.

3.7.2 Model (Simplified Sieve)

Let π be any correct process. We make the following assumptions about an adversarial execution of Simplified Sieve:

- As we established in Section 1.3, the cob adversary does not know which correct processes are in π's echo samples. The cob adversary knows, however, which Byzantine processes are in π's echo samples.
- At any time, the cob adversary knows if π delivered a message. If π delivered a message, then the cob adversary knows which message did π deliver. Moreover, if π delivered a message m, then at any time the cob adversary also knows the processes in π's echo sample for m that sent an Echo(m, m') message to π, for some message m'.
- The cob adversary can cause π to pb.Deliver any message. As we established with Theorem 1, π will, however, pb.Deliver only one message throughout an execution of Simplified Sieve.

Throughout an adversarial execution of Simplified Sieve, an adversary performs a sequence of minimal operations on the system. Each operation consists of either of the following:

- Selecting a correct process that did not pb.Deliver any message, and causing it to pb.Deliver a message.
- Selecting a Byzantine process and causing it to send an Echo message to a correct process.

As a result of each operation, zero or more correct processes deliver a message. The cob adversary is successful if, at the end of the adversarial execution, at least two different messages are delivered by at least one correct process.

3.7.3 Network scheduling

In this section, we discuss the behavior of the adversary in relation to network scheduling. As we discussed in Section 1.3, the system is asynchronous, i.e., every message is eventually delivered but can be delayed by an arbitrary, finite amount of time.

`Gossip` **messages.** As we stated in Section 3.7, throughout this section we assume that the pb instance used by Sieve and Simplified Sieve satisfies totality. While this means that the adversary cannot prevent any correct process from eventually pb.Delivering a message, the adversary can indeed arbitrarily choose which correct process pb.Delivers which message.

This can be achieved by delaying the delivery of the `Gossip` messages issued by correct processes. Noting that a correct process will accept a `Gossip` message from any source, the adversary can then cause any of the processes it controls to quickly send a `Gossip` message with arbitrary content to any correct process, effectively causing it to pb.Deliver an arbitrary message.

`Echo` **messages.** As we stated in Sections 3.7.1 and 3.7.2, the two minimal operations a (pcb or cob) adversary can perform essentially reduce to causing a Byzantine process to either send a `Gossip` or an `Echo` message to a correct process. We can see that those operations are indeed minimal: a correct process atomically al.Delivers a message (i.e., a message is the minimal amount of information that can be meaningfully transferred on the network), and a correct process will ignore any message that is not a `Gossip` or an `Echo` message.

Upon pb.Delivering a message, a correct process will issue zero or more `Echo` messages. As we discussed in Section 1.3, the adversary can arbitrarily delay those messages, but they will eventually be delivered. As a result, the outcome of an adversarial execution is solely determined by the sequence of operations performed by the adversary, and is not affected by network scheduling.

While the adversary could delay the delivery of `Echo` messages issued by correct processes, the only effect this would have is to prevent the adversary from knowing the effect of an operation on the system before performing the next one. An optimal adversary, therefore, performs an operation, then waits until all the `Echo` messages issued by correct processes are delivered before performing the next operation.

3.7.4 Interfaces

In Sections 3.7.1 and 3.7.2, we defined the model underlying an adversarial execution of Sieve and Simplified Sieve respectively. In Section 3.7.3, we discussed the behavior of the Byzantine adversary in relation to network scheduling. Throughout the rest of this chapter, we concretely model a (pcb or cob) adversary as an *algorithm* that interacts with a *system*.

As we discussed, a (pcb or cob) adversary works in steps: at every step, the adversary either performs one operation on the system, or queries the system for information about its state. In this section, we model this interaction by defining four interfaces, respectively implemented by the (pcb or cob) adversary and the (pcb or cob) system.

Both the **pcb adversary** and the **cob adversary** interfaces (instance *adv*) expose the following **procedures**:

- $Init()$: It is called once, at the beginning of the adversarial execution, before any operation is performed on the system. Here the (pcb or cob) adversary setups its internal state.
- $Step()$: It is called repeatedly, until the adversarial execution is completed. Here the (pcb or cob) adversary performs one operation on the system. The execution fails (e.g., an exception is raised) if a call to $adv.Step()$ does not result in one, and only one, call to $sys.Deliver(\ldots)$, $sys.Echo(\ldots)$ or $sys.End()$ (as we define them below).

The **pcb system** interface (instance sys) exposes the following **procedures**:

- $Byzantine(process \in \Pi_C)$: Returns a list of all the Byzantine processes in $process$' echo sample. The pcb adversary can invoke this procedure an unlimited number of times both from the $Init()$ and the $Step()$ procedure.
- $State()$: Returns a list of pairs $(\pi \in \Pi_C, m \in \mathcal{M})$, representing which correct process currently delivered which message. The pcb adversary can invoke this procedure an unlimited number of times from the $Step()$ procedure.
- $Deliver(process \in \Pi_C, message \in \mathcal{M})$: Causes $process$ to pb.Deliver $message$. The execution fails if $Deliver$ is provided with the same $process$ argument more than once: a correct process does not pb.Deliver more than one message. The procedure does not return any value.
- $Echo(process \in \Pi_C, source \in \Pi \setminus \Pi_C, message \in \mathcal{M})$: Causes $source$ to send an `Echo`$(message)$ message to $process$. The execution fails if $Echo$ is provided with the same $process$ and $source$ arguments more than once: a correct process does not consider more than one `Echo` message from the same source. The procedure does not return any value.
- $End()$: Causes the execution to gracefully terminate. The execution fails if $End()$ is called before $Deliver(\ldots)$ is invoked exactly C times: under the assumption that pb satisfies totality, every correct process eventually pb.Delivers a message. The procedure does not return any value.

The **cob system** interface (instance sys) exposes the following **procedures**:

- $Byzantine(process \in \Pi_C)$: Returns a list of all the Byzantine processes in the first echo sample of $process$. The cob adversary can invoke this procedure an unlimited number of times both from the $Init()$ and the $Step()$ procedure.

- $State()$: Returns a list of pairs $(\pi \in \Pi_C, m \in \mathcal{M})$, representing which correct process currently delivered which message. The cob adversary can invoke this procedure an unlimited number of times from the $Step()$ procedure.
- $Sample(process \in \Pi_C, message \in \mathcal{M})$: Returns the processes that are in the echo sample for message $message$ of process $process$ and that sent an `Echo`($message$, $message'$) to $process$, for some message $message'$. The cob adversary can invoke this procedure an unlimited number of times from the $Step()$ procedure. The execution fails if no correct process has cob.Delivered $message$: a correct process does not *reveal* its echo sample for $message$ before $message$ is delivered by at least one correct process.
- $Deliver(process \in \Pi_c, message \in \mathcal{M})$: Causes $process$ to pb.Deliver $message$. The execution fails if $Deliver$ is provided with the same $process$ argument more than once: a correct process does not pb.Deliver more than one message. The procedure does not return any value.
- $Echo(process \in \Pi_C, sample \in \mathcal{M}, source \in \Pi \setminus \Pi_C, message \in \mathcal{M})$: Causes $source$ to send an `Echo`($sample$, $message$) message to $process$. The execution fails if, throughout an execution, $Echo$ is provided with the same $process$, $sample$ and $source$ arguments more than once: a correct process does not consider more than one `Echo` message for the same sample from the same source. The procedure does not return any value.
- $End()$: Causes the execution to gracefully terminate. The execution fails if $End()$ is called before $Deliver(\ldots)$ is invoked exactly C times: under the assumption that pb satisfies totality, every correct process eventually pb.Delivers a message. The procedure does not return any value.

3.8 Simplified adversarial power

In this section, we prove that an optimal consistency-only broadcast adversary is more powerful than an optimal probabilistic consistent broadcast adversary. This result is intuitive: a correct process in Simplified Sieve can deliver more than one message, and in general more information is available to the cob adversary than to the pcb adversary.

3.8.1 Preliminary definitions

Before proving that an optimal cob adversary is more powerful than an optimal pcb adversary, we provide some definitions on pcb and cob systems and adversaries.

Definition 3 (Pcb system)**.** A **pcb system** σ is an element of the set

$$\begin{aligned} \mathcal{S}_{pcb} &= \mathcal{E}_{pcb}^C \\ \mathcal{E}_{pcb} &= \Pi^E \end{aligned}$$

Intuitively, a system $\sigma \in \mathscr{S}_{pcb}$ is defined by the echo sample of each of its C correct processes. The echo sample of a correct process is a vector of E processes.

Let $\sigma \in \mathscr{S}_{pcb}$. We use $\sigma[\pi \in \Pi_C][i \in 1..E]$ to denote the i-th process in π's echo sample.

Definition 4 (Cob system)**.** A **cob system** σ is an element of the set

$$\begin{aligned} \mathscr{S}_{cob} &= \mathscr{E}_{cob}^{C} \\ \mathscr{E}_{cob} &= \left\{\left(e_1, \ldots, e_C \in \Pi^E\right) \mid \mathscr{M}\left(e_i, e_j\right)\ \forall i, j \in 1..C\right\} \\ \mathscr{M}(e, e') &: \quad \forall k, (e_k \in \Pi \setminus \Pi_C) \implies \left(e'_k = e_k\right) \end{aligned}$$

Intuitively, a system $\sigma \in \mathscr{S}_{cob}$ is defined by the echo samples of each of its C correct processes. Each correct process has C echo samples $e_1, \ldots, e_C$ (one per message), each represented by a vector of E processes. Any two echo samples e_i, e_j of a given process satisfy $\mathscr{M}(e_i, e_j)$, i.e., they share the same set of Byzantine processes.

We also use just $\mathscr{S}$ to denote the set of cob systems $\mathscr{S}_{cob}$. Let $\sigma \in \mathscr{S}_{cob}$, we use $\sigma[\pi \in \Pi_C][m \in \mathscr{M}][i \in 1..E]$ to denote the i-th process in π's echo sample for m.

Definition 5 (Adversary)**.** A **pcb adversary** (**cob adversary**) is a terminating algorithm that exposes the pcb adversary (cob adversary) interface and does not cause the adversarial execution to fail (see Section 3.7.4) when coupled with any system $\sigma \in \mathscr{S}_{pcb}$ ($\sigma \in \mathscr{S}_{cob}$).

Let α, α' be two pcb (cob) adversaries such that, for every $\sigma \in \mathscr{S}_{pcb}$ ($\sigma \in \mathscr{S}_{cob}$), the execution of α coupled with σ is identical to the execution of α' coupled with σ. We consider α and α' to be functionally the same adversary.

We use $\mathscr{A}_{pcb}$ to denote the set of pcb adversaries. We use $\mathscr{A}_{cob}$ (or just $\mathscr{A}$) to denote the set of cob adversaries.

Definition 6 (Adversarial power)**.** Let α be a pcb (cob) adversary. The **adversarial power** of α is the probability of α compromising the consistency of a pcb (cob) system, picked with uniform probability from $\mathscr{S}_{pcb}$ ($\mathscr{S}_{cob}$).

Definition 7 (Optimal adversary)**.** Let α be a pcb (cob) adversary. We say that α is an **optimal adversary** if its adversarial power is greater or equal to that of any other pcb (cob) adversary.

We note that Definition 7 is well defined: indeed, both $\mathscr{A}_{pcb}$ and $\mathscr{A}_{cob}$ are finite sets, and therefore admit a maximum for the adversarial power.

Definition 8 (Optimal set of adversaries)**.** Let $\mathscr{A}'$ be a set of pcb (cob) adversaries. We say that $\mathscr{A}'$ is an **optimal set of adversaries** if $\mathscr{A}'$ includes an optimal pcb (cob) adversary.

Definition 9 (Pcb invocation/response pair)**.** The pair (i, r) is a **pcb invocation/response pair**

if

$$
\begin{aligned}
i &= (\texttt{Byzantine}, \pi \in \Pi_C) && r = (\xi_1, \ldots, \xi_k \in \Pi \setminus \Pi_C) \\
i &= (\texttt{State}) && r = ((\pi_1, m_1), \ldots, \\
& && \quad (\pi_k \in \Pi_C, m_k \in \mathscr{M})) \\
i &= (\texttt{Deliver}, \pi \in \Pi_C, m \in \mathscr{M}) && r = \bot \\
i &= (\texttt{Echo}, \pi \in \Pi_C, \xi \in \Pi \setminus \Pi_C, m \in \mathscr{M}) && r = \bot
\end{aligned}
$$

Definition 10 (Cob invocation/response pair)**.** The pair (i, r) is a **cob invocation/response pair** if

$$
\begin{aligned}
i &= (\texttt{Byzantine}, \pi \in \Pi_C) && r = (\xi_1, \ldots, \xi_k \in \Pi \setminus \Pi_C) \\
i &= (\texttt{Sample}, \pi \in \Pi_C, m \in \mathscr{M}) && r = (\rho_1, \ldots, \rho_k \in \Pi) \\
i &= (\texttt{State}) && r = ((\pi_1, m_1), \ldots, \\
& && \quad (\pi_k \in \Pi_C, m_k \in \mathscr{M})) \\
i &= (\texttt{Deliver}, \pi \in \Pi_C, m \in \mathscr{M}) && r = \bot \\
i &= (\texttt{Echo}, \pi \in \Pi_C, s \in \mathscr{M}, \xi \in \Pi \setminus \Pi_C, m \in \mathscr{M}) && r = \bot
\end{aligned}
$$

Definition 11 (Trace)**.** A **pcb trace** (**cob trace**) is a finite sequence of pcb (cob) invocation/response pairs. Let α be a (pcb or cob) adversary, let σ be a (pcb or cob, correspondingly) system. We use $\tau(\alpha, \sigma)$ to denote the trace produced by α coupled with σ. We use $\mathscr{T}$ to denote the set of traces.

Notation 5 (Power set)**.** Let X be a set. We use $\mathbb{P}(X)$ to denote the **power set** of X. We use $\mathbb{P}^K(X) = \{x \in \mathbb{P}(X) \mid |x| = K\}$ to denote the elements in $\mathbb{P}(X)$ that have K elements. We use $\mathbb{P}^{K+}(X) = \{x \in \mathbb{P}(X) \mid |x| \geq K\}$ to denote the elements in $\mathbb{P}(X)$ that have at least K elements.

3.8.2 Consistency of Simplified Sieve

We can now prove that the ϵ-consistency of Simplified Sieve is strictly weaker than that of Sieve.

Lemma 12. *An optimal cob adversary is more powerful than an optimal pcb adversary.*

Proof. Let α^* be an optimal pcb adversary. In order to prove that an optimal cob adversary is more powerful than α^*, we just need to find a cob adversary α^+ that is more powerful than α^*. We achieve this using a **pcb-to-cob decorator**, i.e., an algorithm that acts as an interface between a pcb adversary and cob system. A pcb adversary coupled with a pcb-to-cob decorator effectively implements a cob adversary. Here we show that a pcb-to-cob decorator Δ_{cob} exists such that, for every $\alpha \in \mathscr{A}_{pcb}$, the cob adversary $\alpha' = \Delta_{cob}(a)$ is more powerful than α. If this is true, the lemma is proved: indeed, $\alpha^+ = \Delta_{cob}(\alpha^*)$ is more powerful than α^*.

Algorithm 6 `Cob decorator.`

```
1:  Implements:
2:      CobAdversary + PcbSystem, instance cadv
3:
4:  Uses:
5:      PcbAdversary, instance padv, system cadv
6:      CobSystem, instance sys
7:
8:  procedure cadv.Init() is
9:      deliveries = {⊥}^C;
10:     padv.Init();
11:
12: procedure cadv.Step() is
13:     padv.Step();
14:
15: procedure cadv.Byzantine(process) is
16:     return sys.Byzantine(process);
17:
18: procedure cadv.State() is
19:     state = ∅;
20:
21:     for all (π, m) ∈ sys.State() do
22:         if deliveries[π] = m then
23:             state ← state ∪ {(π, m)};
24:         end if
25:     end for
26:
27:     return state;
28:
29: procedure cadv.Deliver(process, message) is
30:     deliveries[process] ← message;
31:     sys.Deliver(process, message);
32:
33: procedure cadv.Echo(process, source, message) is
34:     sys.Echo(process, message, source, message);
35:
36: procedure cadv.End() is
37:     sys.End();
38:
```

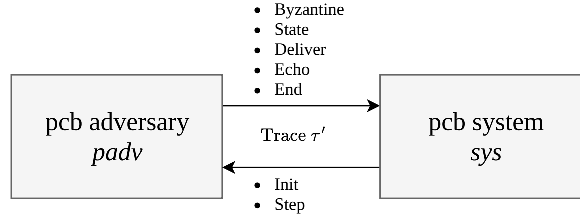


Figure 3.1: An execution without decorator.

Decorator. Algorithm 6 implements `Cob decorator`, a pcb-to-cob decorator. Provided with a pcb adversary $padv$, `Cob decorator` acts as an interface between $padv$ and a cob system sys, effectively implementing a cob adversary $cadv$. `Cob decorator` exposes both the cob adversary and the pcb system interfaces: the underlying pcb adversary $padv$ uses $cadv$ as its system.

`Cob decorator` works as follows:

- Procedure $cadv.Init()$ initializes a $deliveries$ array that is used to keep track of the message pb.Delivered by each correct process, and a gap set that it uses to keep track of the messages cob.Delivered by each correct process in sys.
- Procedure $cadv.Step()$ simply forwards the call to $padv.Step()$.
- Procedure $cadv.State()$ returns a list of pairs $(\pi \in \Pi_C, m \in \mathcal{M})$ such that π both pb.Delivered and delivered m in sys. This is achieved by querying $sys.State()$, then looping over each element (π, m) of the response and checking if $deliveries[\pi] = m$.
- Procedure $cadv.Byzantine(process)$ simply forwards the call to $sys.Byzantine(process)$.
- Procedure $cadv.Deliver(process, message)$ sets $deliveries[process]$ to $message$ (to signify that $process$ pb.Delivered $message$). It then forwards the call to $sys.Deliver(process, message)$, causing $process$ to pb.Deliver $message$.
- Procedure $cadv.Echo(process, source, message)$ forwards the call to $sys.Echo(process, message, source, message)$, causing $source$ to send an `Echo`($message$, $message$) message to $process$.
- Procedure $cadv.End()$ simply forwards the call to $sys.End()$.

Let $\Delta_{cob} : \mathscr{A}_{pcb} \to \mathscr{A}_{cob}$ denote the function that `Cob decorator` implements, mapping pcb adversaries into cob adversaries. We want to prove that, for every $\alpha \in \mathscr{A}_{pcb}$, the adversarial power of $\alpha' = \Delta_{cob}(\alpha)$ is greater than that of α.

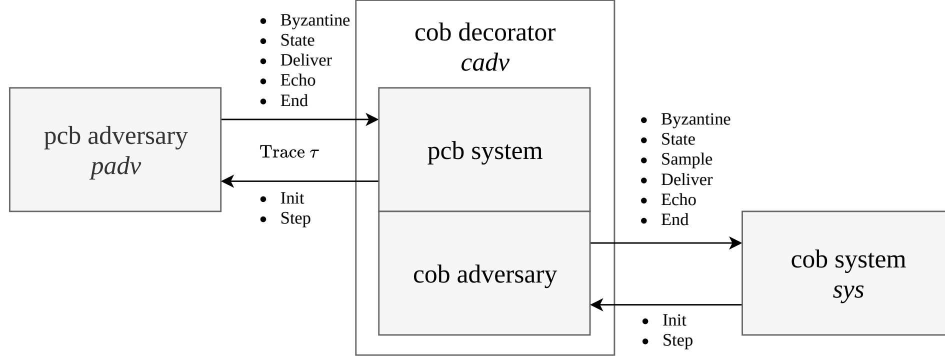


Figure 3.2: A decorator exposing both its system interface to the pcb adversary and its adversary interface to the cob system.

System translation. Let α be a pcb adversary. We start by noting that, since α is correct, α always causes every correct process to pb.Deliver a message. We can therefore define a function

$$\mu : \mathscr{A}_{pcb} \times \mathscr{S}_{pcb} \times \Pi_C \to \mathscr{M}$$

such that $\mu(\alpha, \sigma, \pi) = m$ if and only if α eventually causes π to pb.Deliver m, when α is coupled with σ.

We then define a **system translation** function $\Psi[\alpha] : \mathscr{S}_{pcb} \to \mathbb{P}(\mathscr{S}_{cob})$ that maps a pcb system into a set of cob systems:

$$\left(\sigma' \in \Psi[\alpha](\sigma)\right) \iff \left(\forall \pi \in \Pi_C, \sigma[\pi] = \sigma'[\pi][\mu(\alpha, \sigma, \pi)]\right)$$

Let σ be a pcb system, let σ' be a cob system, let π be any correct process, let m be the message that α eventually causes π to pb.Deliver, when α is coupled with σ. Intuitively, σ' is in $\Psi[\alpha](\sigma)$ if π's echo sample for m in σ' is identical to π's echo sample in σ.

Roadmap. Let $\alpha \in \mathscr{A}_{pcb}$, $\alpha' = \Delta_{cob}(\alpha)$. Let $\sigma \in \mathscr{S}_{pcb}$ such that α compromises the consistency of σ. In order to prove that α' is more powerful than α, we prove that:

- For every $\sigma' \in \Psi[\alpha](\sigma)$, α' compromises the consistency of σ'.
- The probability of $\Psi[\alpha](\sigma)$ is equal to the probability of σ.
- For every $\hat{\sigma} \in \mathscr{S}_{pcb}$ such that $\hat{\sigma} \neq \sigma$, the sets $\Psi[\alpha](\sigma)$ and $\Psi[\alpha](\hat{\sigma})$ are disjoint.

Indeed, if all of the above are true, then the probability of α' compromising the consistency of a random cob system σ' is greater or equal to the probability of α compromising the consistency of a random system σ, and the lemma is proved.

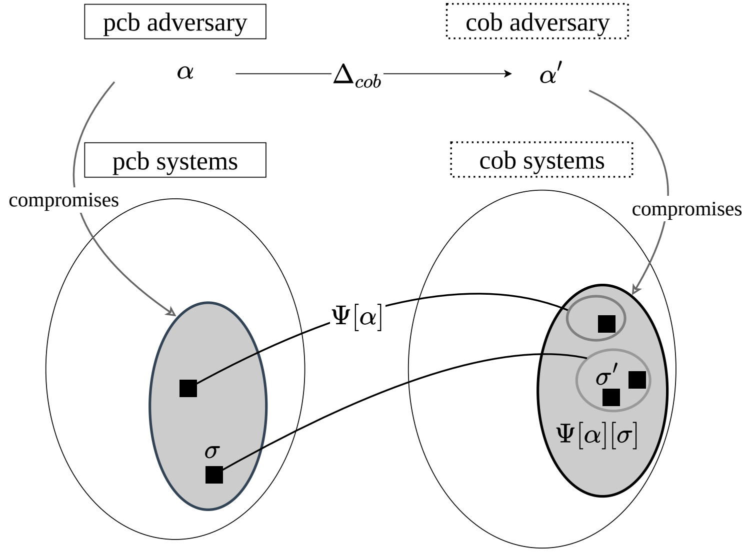


Figure 3.3: An illustration of the steps needed to prove that the adversarial power of α is greater than that of α'.

Trace. We start by noting that, if we couple `Cob decorator` with σ', we effectively obtain a pcb system interface δ with which α directly exchanges invocations and responses. Here we show that the trace $\tau(\alpha,\sigma)$ is identical to the trace $\tau(\alpha,\delta)$. Intuitively, this means that α has no way of *distinguishing* whether it has been coupled directly with σ, or it has been coupled with σ', with `Cob decorator` acting as an interface. We prove this by induction.

Let us assume

$$\begin{aligned} \tau(\alpha,\sigma) &= ((i_1, r_1), \ldots) \\ \tau(\alpha,\delta) &= ((i'_1, r'_1), \ldots) \\ i_j = i'_j, r_j = r'_j & \qquad \forall j \leq n \end{aligned}$$

with $n \geq 0$ (here $n = 0$ means that this is α's first invocation). We start by noting that, since a is a deterministic algorithm, we immediately have

$$i_{n+1} = i'_{n+1}$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Byzantine}, \pi)$. By hypothesis, at least one of the echo samples of π in σ' is identical to the echo sample of π in σ. Moreover, all π's echo samples in σ' share the same set of Byzantine processes. Therefore, the first of π's echo samples in σ' contains the same Byzantine processes as π's echo sample in σ. Finally, the decorator simply forwards the call to $cadv.Byzantine(\pi)$ to $sys.Byzantine(\pi)$. Consequently, $r_{n+1} = r'_{n+1}$.

Before considering the case $i_{n+1} = (\texttt{State})$, we prove some auxiliary results. Let π be a correct process, let ρ be a process, let ξ be a Byzantine process, let m be a message. For every $j \leq n+1$, as we established, we have $i_j = i'_j$. Therefore, after the $(n+1)$-th invocation, the following hold true:

- π pb.Delivered m in σ' if and only if π pb.Delivered m in σ. Indeed, $cadv.Deliver(\pi, m)$ was invoked if and only if $sys.Deliver(\pi, m)$ was invoked as well.
- π pb.Delivered m in σ' if and only if $deliveries[\pi] = m$. Indeed, $cadv.Deliver(\pi, m)$ was invoked if and only if $deliveries[\pi]$ was set to m.
- ξ sent an $\texttt{Echo}(m)$ to π in σ' if and only if ξ sent an $\texttt{Echo}(m,\ m)$ message to π in σ. Indeed, $cadv.Echo(\pi, \xi, m)$ was invoked if and only if $sys.Echo(\pi, m, \xi, m)$ was invoked as well.
- If π pb.Delivered m in σ, then π's echo sample for m in σ' is identical to π's echo sample in σ. This follows from the definition of Ψ (we recall that $\sigma' \in \Psi[\alpha](\sigma)$).
- If π delivered m in σ, it also delivered m in σ'. Indeed, since π pb.Delivered m in σ, π's echo sample for m in σ' is identical to π's echo sample in σ. Moreover, if π received an $\texttt{Echo}(m)$ message from ρ in σ, then it also received an $\texttt{Echo}(m,\ m)$ message from ρ in σ'.
- If π both pb.Delivered and delivered m in σ', it also delivered m in σ. Indeed, since π pb.Delivered m in σ', then it also pb.Delivered m in σ, and π's echo sample in σ is identical to π's echo sample for m in σ'. Moreover, if π received an $\texttt{Echo}(m)$ message from ρ in σ', then it also received an $\texttt{Echo}(m,\ m)$ message from ρ in σ.

Let us assume $i_{n+1} = (\texttt{State})$. We start by noting that $cadv.State()$ returns all the pairs (π', m') in $sys.State()$ that satisfy $deliveries[\pi'] = m'$. If $(\pi, m) \in r_{n+1}$, then π both pb.Delivered and delivered m both in σ. Therefore, π both pb.Delivered and delivered m in σ', and $deliveries[\pi] = m$. Consequently, $(\pi, m) \in r'_{n+1}$. If $(\pi, m) \in r'_{n+1}$, then (π, m) was returned from $sys.State()$, and $deliveries[\pi] = m$. Therefore, π both pb.Delivered and delivered m in σ'. Consequently, π delivered m in σ, and $(\pi, m) \in r_{n+1}$.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \bot = r'_{n+1}$. By induction, we have $\tau(\alpha, \sigma) = \tau(\alpha, \delta)$.

Consistency of σ'. We proved that $\tau(\alpha, \sigma) = \tau(\alpha, \delta)$. Moreover, we proved that if a correct process π eventually pcb.Delivers a message m in σ, then π also cob.Delivers m in σ'.

Since α compromises the consistency of σ, two correct processes π, π' and two distinct messages m, $m' \neq m$ exist such that, in σ, π pcb.Delivered m and π' pcb.Delivered m'. Therefore, in σ', π cob.Delivered m and π' cob.Delivered m'. Therefore α' compromises the consistency of σ'.

Translation probabilities. We now prove that, for every $\sigma \in \mathscr{S}_{pcb}$, the probability of $\Psi[\alpha](\sigma)$ is equal to the probability of σ.

The probability of σ is

$$\mathscr{P}[\sigma] = \mathscr{P}\big[\sigma[\pi_1][1] = \pi_{1,1}, \ldots, \sigma[\pi_C][E] = \pi_{C,E}\big] = N^{-EC}$$

and the probability of $\Psi[\alpha](\sigma)$ is

$$\begin{array}{rcl} \mathscr{P}[\Psi[\alpha](\sigma)] & = & \\ \mathscr{P}\big[\sigma[\pi_1][\mu(\alpha,\sigma,\pi_1)][1] = \pi_{1,1}, \ldots, \sigma[\pi_C][\mu(\alpha,\sigma,\pi_C)[E] = \pi_{C,E}]\big] & = & N^{-EC} \end{array}$$

which proves the result.

Translation disjunction. We now prove that, for any two σ_a, $\sigma_b \neq \sigma_a$, we have $\Psi[\alpha](\sigma_a) \cap \Psi[\alpha](\sigma_b) = \emptyset$. We prove this by contradiction. Suppose a system σ' exists such that $\sigma' \in \Psi[\alpha](\sigma_a)$ and $\sigma' \in \Psi[\alpha](\sigma_b)$. We want to prove that $\sigma_a = \sigma_b$.

We start by noting that, if $\tau(\alpha,\sigma_a) = \tau(\alpha,\sigma_b)$, then $\sigma_a = \sigma_b$. Indeed, we have

$$\begin{array}{rcl} \tau(\alpha,\sigma_a) & = & \tau(\alpha,\sigma_b) \\ \implies \mu(\alpha,\sigma_a,\pi) & = & \mu(\alpha,\sigma_b,\pi)\ \forall \pi \in \Pi_C \\ \implies \sigma_a[\pi] & = & \sigma'[\pi][\mu(\alpha,\sigma_a,\pi)] \\ & = & \sigma'[\pi][\mu(\alpha,\sigma_b,\pi)] \\ & = & \sigma_b[\pi]\ \forall \pi \\ \implies \sigma_a & = & \sigma_b \end{array}$$

We prove that $\tau(\alpha,\sigma_a) = \tau(\alpha,\sigma_b)$ by induction. Let us assume

$$\begin{array}{rcl} \tau(\alpha,\sigma_a) & = & ((i_1, r_1), \ldots) \\ \tau(\alpha,\sigma_b) & = & ((i'_1, r'_1), \ldots) \\ i_j = i'_j, r_j = r'_j & & \forall j \leq n \end{array}$$

with $n \geq 0$ (here $n = 0$ means that this is α's first invocation). We start by noting that, since a is

a deterministic algorithm, we immediately have

$$i_{n+1} = i'_{n+1}$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Byzantine}, \pi)$. By hypothesis, among the echo samples of π in σ', at least one is identical to the echo sample of π in σ_a, and at least one is identical to the echo sample of π in σ_b. Noting that π's echo samples share the same set of Byzantine processes, we immediately have that the Byzantine processes in $\sigma_a[\pi]$ are the same as in $\sigma_b[\pi]$, and $r_{n+1} = r'_{n+1}$.

Before considering the case $i_{n+1} = (\texttt{State})$, we prove some auxiliary results. Let π be a correct process, let ρ be a process, let ξ be a Byzantine process, let m be a message. For every $j \leq n+1$, as we established, we have $i_j = i'_j$. Therefore, after the $(n+1)$-th invocation, the following hold true:

- π pb.Delivered m in σ_a if and only if π pb.Delivered m in σ_b.
- ξ sent an $\texttt{Echo}(m)$ message to π in σ_a if and only if ξ send an $\texttt{Echo}(m)$ message to π in σ_b.
- If π pb.Delivered m (both in σ_a and σ_b), then $\sigma_a[\pi] = \sigma_b[\pi]$. Indeed,

$$\begin{aligned} \sigma_a[\pi] &= \sigma'[\pi][\mu(\alpha, \sigma_a, \pi)] \\ &= \sigma'[\pi][\mu(\alpha, \sigma_b, \pi)] \\ &= \sigma_b[\pi] \end{aligned}$$

- π delivered m in σ_a if and only if π delivered m in σ_b. Indeed, if π delivered m in σ_a, then it also pb.Delivered m in σ_a and, consequently, σ_b. Therefore, π's echo sample in σ_a is identical to π's echo sample in σ_b. Since π received the same $\texttt{Echo}$ messages in σ_a and σ_b then π delivered m in σ_b. The argument can be trivially reversed to prove that, if π delivered m in σ_b, then π also delivered m in σ_a.

Let us consider the case $i_{n+1} = (\texttt{State})$. From the above follows $r_{n+1} = r'_{n+1}$.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \bot = r'_{n+1}$. By induction, we have $\tau(\alpha, \sigma_a) = \tau(\alpha, \sigma_b)$.

Therefore, $\sigma_a = \sigma_b$, which contradicts the hypothesis and thus proves that the sets $\Psi[\alpha](\sigma_a)$ and $\Psi[\alpha](\sigma_b)$ are disjoint.

□

3.9 Two-phase adversaries

In Section 3.8 we proved the important result that it is easier to compromise the consistency of Simplified Sieve than that of Sieve. Throughout the rest of this chapter, we compute a bound on the ϵ-security of Simplified Sieve.

It is easy to see that the ϵ-security of Simplified Sieve is equal to the adversarial power of an optimal adversary. Therefore, ϵ_c is a bound on the ϵ-security of Simplified Sieve if ϵ_c bounds the adversarial power of every adversary in an optimal set of adversaries.

In this section, we derive a set $\mathscr{A}_{tp} \subseteq \mathscr{A}$ of *two-phase* adversaries that we prove to be optimal. Unlike $\mathscr{A}$, $\mathscr{A}_{tp}$ is small enough to be probabilistically tractable. In the next sections, we compute a bound on the adversarial power of every $a \in \mathscr{A}_{tp}$.

In a similar way to Lemma 12, the proofs of optimality of most of the sets of adversaries presented in this section make extensive use of decorators, and are in general lengthy and non-trivial. For the sake of readability, in this section we only state our results, and defer each explicit proof to Section 3.11.

Auto-echo adversary

As we introduced in Section 3.6.3, an Echo message in Simplified Sieve has two fields: a sample s and a message m. Intuitively, an Echo(s, m) message represents the following statement: “within the context of message s, consider my Echo to be for message m".

Upon pb.Delivering a message m, a correct process sends to every other process an Echo(s, m) for every s. In other words, a correct process supports the message it pb.Delivers across all samples. A Byzantine process, however, is not constrained to do this.

A correct process cob.Delivers a message m upon collecting enough Echo(m, m) messages from its echo sample for m. It is easy to see, therefore, that the probability of a correct process π cob.Delivering m increases if all the Byzantine processes send an Echo(m, m) message to π.

Definition 12 (Auto-echo adversary)**.** An adversary $a \in \mathscr{A}$ is an **auto-echo adversary** if, at the beginning of its execution, it causes ξ to send an Echo(m, m) message to π, for every $\pi \in \Pi_c$, $\xi \in \Pi \setminus \Pi_C$ and $m \in \mathscr{M}$. We use $\mathscr{A}_{ae}$ to denote the set of auto-echo adversaries.

In Section 3.11.1, we formally prove this intuition, i.e., we prove that the set of auto-echo adversaries $\mathscr{A}_{ae}$ is optimal.

Process-sequential adversary

As we discussed in Section 3.6.3, a correct process reveals its sample for a message m only after delivering m. At the beginning of the execution, the adversary only knows which Byzantine

processes are in each correct process' echo samples. In Section 3.9, however, we proved that this does not affect the optimal adversary's strategy: the set of Byzantine processes in a correct process' echo samples don't play any role in an optimal adversarial execution.

Intuitively, therefore, an optimal adversary has effectively no meaningful way to distinguish any two correct processes based on the outcome that their actions will have on the system.

Definition 13 (Correct process enumeration)**.** We define a bijection

$$\zeta : 1..C \leftrightarrow \Pi_C$$

that uniquely maps an integer identifier $i \in 1..C$ to a correct process.

Definition 14 (Process-sequential adversary)**.** An auto-echo adversary $\alpha \in \mathscr{A}_{ae}$ is a **process-sequential adversary** if it never causes $\zeta(i)$ to pb.Deliver a message before any $\zeta(j < i)$. We use $\mathscr{A}_{ps}$ to denote the set of process-sequential adversaries.

In Section 3.11.2, we formally prove this intuition, i.e., we prove that the set of process-sequential adversaries $\mathscr{A}_{ps}$ is optimal.

Sequential adversary

As we introduced in Section 3.6.3, in Simplified Sieve a correct process independently selects C echo samples, one for every message in $\mathscr{M}$. Moreover, every echo sample shares the same set of Byzantine processes. Finally, let π be a correct process, let m be a message, no correct process in π's echo sample for m is known to the adversary before π delivers m.

Intuitively, therefore, an adversary has effectively no meaningful way of distinguishing two messages, based on the outcome that their pb.Delivery will have on the system.

Definition 15 (Poisoned process)**.** Let σ be a system, let π be a correct process. We say that π **is poisoned in** σ if and only if at least $\hat{E}$ processes in π's first echo sample in σ are Byzantine.

Definition 16 (Sequential adversary)**.** A process-sequential adversary $\alpha \in \mathscr{A}_{ps}$ is a **sequential adversary** if it never causes a correct process to pb.Deliver $m \in \mathscr{M}$ before causing every $l < m \in \mathscr{M}$ to be pb.Delivered by at least one correct process. We use $\mathscr{A}_{sq}$ to denote the set of sequential adversaries.

In Section 3.11.3, we formally prove this intuition, i.e., we prove that the set of sequential adversaries $\mathscr{A}_{sq}$ is optimal.

Non-redundant adversary

As we established in Section 3.6.1, the consistency of consistency-only broadcast is compromised if and only if at least two messages are delivered by at least one correct process.

It is easy to see, therefore, that an adversary that has already caused at least one correct process to deliver a message m gains no advantage from causing more correct processes to pb.Deliver m. Indeed, doing so would not increase the probability of at least one correct process delivering m (that condition is verified with probability 1): an optimal adversary should focus its remaining pb.Deliveries on achieving the goal to cause at least one other message to be delivered by at least one correct process.

Definition 17 (Non-redundant adversary)**.** A sequential adversary $\alpha \in \mathscr{A}_{sq}$ is a **non-redundant adversary** if, whenever exactly one message m has been delivered, it never causes any additional correct process to pb.Deliver m. We use $\mathscr{A}_{nr}$ to denote the set of non-redundant adversaries.

In Section 3.11.4, we formally prove this intuition, i.e., we prove that the set of non-redundant adversaries $\mathscr{A}_{nr}$ is optimal.

Sample-blind adversary

In Section 3.6.3, we discussed how, in Simplified Sieve, a correct process reveals its echo sample for a message after at least one correct process delivered that message. Throughout Section 3.10.1, we extensively used `Reveal` messages (through the $Sample(\ldots)$ system interface) to build a sequence of decorators that improved the power of any adversary in their domain.

In this section, we prove the counter-intuitive result that the information contained in a `Reveal` message is actually useless to an optimal adversary. Indeed, the decorators we developed leveraged `Reveal` messages to *correct* the sub-optimal behavior of a generic adversary. However, for every decorator that we developed, we argue that we could develop an adversary in the codomain of that decorator that never uses the information provided by `Reveal` messages.

An intuitive insight on `Reveal` messages can be provided by the observation that the information they provide is disclosed in the moment it ceases to actually be useful. Indeed, a correct process reveals the content of its echo sample for a message m only after at least one correct process delivered m. As we proved in Section 3.9, causing additional processes to deliver m gives no advantage to the adversary. Moreover, since the correct processes in each echo sample are picked independently from each other, the knowledge of a correct process π's echo sample for m does not grant any advantage in causing π to deliver $m' \neq m$.

Notation 6 (Undefined minima and maxima)**.** Let $X \subset \mathbb{N}$, with X finite, let $S : X \to \{\texttt{True}, \texttt{False}\}$ be a predicate on X. We use

$$\begin{aligned}(\min n \in X \mid S(n)) &= \bot \\ (\max n \in X \mid S(n)) &= \bot\end{aligned}$$

to denote that

$$\nexists n \in X \mid S(n)$$

Definition 18 (Trace compatibility)**.** Let τ be a trace, let σ be a system. We say that τ **is compatible with** σ, or $\tau \sim \sigma$, if the sequence of invocations in τ, applied in order to σ, produces the corresponding sequence of responses in τ.

Notation 7 (Consistency compromission)**.** Let α be an adversary, let σ be a system, let τ be a trace. We use $\alpha \searrow \sigma$ to signify that α compromises the consistency of σ. We use $\tau \searrow \sigma$ to signify that the sequence of invocations in τ compromises the consistency of σ.

Definition 19 (Sample-blind adversary)**.** A non-redundant adversary $\alpha \in \mathscr{A}_{nr}$ is a **sample-blind adversary** if it never invokes $Sample(\ldots)$. We use $\mathscr{A}_{sb}$ to denote the set of sample-blind adversaries.

In Section 3.11.5, we formally prove this intuition, i.e., prove that the set of sample-blind adversaries $\mathscr{A}_{sb}$ is optimal.

Byzantine-counting adversary

In Section 3.7.2 we discussed how an adversary for Simplified Sieve knows which Byzantine processes are in the first echo sample of any correct process. In Section 3.9, however, we proved that the optimal adversarial behavior with respect to `Echo` messages is always to cause every Byzantine process to send an `Echo`(m, m) message to every correct process, for every message $m \in \mathscr{M}$.

Intuitively, therefore, a correct process gains no advantage from knowing specifically which Byzantine processes are in the first echo sample of any correct process.

Definition 20 (Byzantine-counting adversary)**.** A sample-blind adversary $\alpha \in \mathscr{A}_{sb}$ is a **Byzantine-counting adversary** if, whenever it invokes $Byzantine(\pi \in \Pi_C)$, it invokes $|Byzantine(\pi)|$. In other words, the behavior of a Byzantine-counting adversary does not depend on the specific set of Byzantine processes in the first echo sample of any correct process. We use $\mathscr{A}_{bc}$ to denote the set of Byzantine-counting adversaries.

In Section 3.11.6, we formally prove this intuition, i.e., we prove that the set of Byzantine-counting adversaries $\mathscr{A}_{bc}$ is optimal.

Single-response adversary

As we introduced in Section 3.7.2, the goal of a cob adversary is to compromise the consistency of a cob system by causing two distinct messages to be delivered by at least one correct process each. In order to achieve this, it acts upon the system in steps, causing correct processes to pb.Deliver a sequence of messages, until the consistency is compromised.

We distinguish two phases of an adversarial execution.

Definition 21 (Trace phases)**.** Let α be an adversary, let σ be a system. We call **first phase of** $\tau(\alpha,\sigma)$ the sequence $\tau(\alpha,\sigma)_1,\ldots,\tau(\alpha,\sigma)_n$ with n given by

$$\begin{aligned} n &= \begin{cases} \min_j \mid S(j) & \text{iff } \exists j \mid S(j) \\ |\tau(\alpha,\sigma)| & \text{otherwise} \end{cases} \\ S(j) &= \left(\tau(\alpha,\sigma)_j = (\texttt{State}, r_h),\ r_h \neq \emptyset\right) \end{aligned}$$

We call $\tau(\alpha,\sigma)_{n+1},\ldots,\tau(\alpha,\sigma)_{|\tau(\alpha,\sigma)|}$ the **second phase of** $\tau(\alpha,\sigma)$. We call $\eta(\alpha,\sigma)$ the first phase of $\tau(\alpha,\sigma)$. We call $\theta(\alpha,\sigma)$ the second phase of $\tau(\alpha,\sigma)$.

The first phase of a trace ends when, for the first time, a call to $State()$ returns a non-empty set. Intuitively, the first phase ends when the adversary becomes aware that at least one correct process delivered a message.

Let us focus on the second phase of an adversarial execution carried out by a Byzantine-counting adversary. We know that, at the beginning of the second phase, at least one message has been delivered by at least one correct process. If more than one message has been delivered, the adversary already compromised the consistency of the system, and the invocations in the second phase are irrelevant to its success.

If exactly one message has been delivered, an optimal adversary will issue a sequence of invocations that, given the information available on the system, maximizes the probability of at least one more message being delivered by at least one correct process. Since the adversary is non-redundant, the response provided by any invocation to $State()$ will not change until the consistency is compromised. Intuitively, therefore, the information available to the adversary throughout the second phase does not change until consistency is compromised. Since any invocation issued by the adversary after consistency is compromised is irrelevant to its success, an optimal adversary does not need to invoke $State()$ throughout the second phase of any adversarial execution.

Definition 22 (Single-response adversary)**.** A Byzantine-counting adversary $\alpha \in \mathscr{A}_{bc}$ is a **single-response adversary** if it never invokes $State()$ throughout the second phase of any adversarial execution. We use $\mathscr{A}_{sr}$ to denote the set of single-response adversaries.

In Section 3.11.7, we formally prove this intuition, i.e., we prove that the set of single-response adversaries $\mathscr{A}_{sr}$ is optimal.

State-polling adversary

In Section 3.9, we proved that an optimal adversary does not need to invoke $State()$ in the second phase of an adversarial execution, i.e., after at least one message has been delivered by

at least one correct process.

It is easy to see, however, that, throughout the first phase, the information provided by $State()$ is useful to the adversary. Intuitively, the sooner a single-response adversary becomes aware that at least one correct process delivered a message, the sooner it can focus its strategy to cause the delivery of a second, distinct message.

In this section, we prove this intuition, i.e., we formally prove that the set of *state-polling adversaries* is optimal.

Definition 23 (State-polling adversary)**.** A single-response adversary $\alpha \in \mathscr{A}_{sr}$ is a **state-polling adversary** if it invokes $State()$ before the first invocation of $Deliver(\ldots)$ and after each invocation of $Deliver(\ldots)$, until $State()$ returns a non-empty set. We use $\mathscr{A}_{sp}$ to denote the set of state-polling adversaries.

Lemma 13. *The set of state-polling adversaries $\mathscr{A}_{sp}$ is optimal.*

Proof. It follows immediately from the observation that, for any adversary, not invoking $State()$ is equivalent to invoking $State()$ and ignoring its response. □

Two-phase adversary

In Section 3.9, we proved that: throughout the first phase, an optimal adversary invokes $State()$ before the first invocation of $Deliver(\ldots)$ and after each invocation of $Deliver(\ldots)$; throughout the second phase, an optimal adversary never needs to invoke $State()$.

As we discussed, if the first phase is concluded with more than one message being delivered, the adversary already compromised the consistency of the system, and the invocations in the second phase are irrelevant to its success.

Let us consider the case where, at the beginning of the second phase, exactly one message m^* has been delivered. In Section 3.6.3, we discussed how a correct process selects the correct component of each echo sample independently. Intuitively, therefore, the knowledge of which processes delivered m^* is useless to the adversary, as it provides no information about the correct component of any echo sample for a message $m \neq m^*$. In other words, an optimal adversary only needs to know *when* the first phase of the execution is concluded, but not *how*.

Definition 24 (Two-phase adversary)**.** A state-polling adversary $\alpha \in \mathscr{A}_{sp}$ is a **two-phase adversary** if, whenever it invokes $State()$, it only invokes $(State() \neq \emptyset)$. In other words, the behavior of a two-phase adversary does not depend on the content of $State()$, but only on whether or not $State()$ is empty.

Lemma 14. *Let α be a state-polling adversary, let σ, σ' be systems such that*

$$\left|\eta(\alpha, \sigma)\right| = \left|\eta(\alpha, \sigma')\right|$$

and, for all $\pi \in \Pi_C$, $m \in \mathcal{M}$,

$$|\{n \in 1..E \mid \sigma[\pi][m][n] \in \Pi_C\}| = \left|\{n \in 1..E \mid \sigma'[\pi][m][n] \in \Pi_C\}\right|$$

We have

$$\forall n < \left|\eta(\alpha,\sigma)\right|, \tau(\alpha,\sigma)_n = \tau(\alpha,\sigma')_n$$

Proof. The lemma is proved by induction. Let us assume

$$\begin{aligned} \tau(\alpha,\sigma) &= ((i_1, r_1), \ldots) \\ \tau(\alpha,\sigma') &= ((i'_1, r'_1), \ldots) \\ i_j = i'_j, r_j = r'_j & \qquad \forall j \leq n \end{aligned}$$

We start by noting that, since α is a deterministic algorithm, we immediately have

$$i_{n+1} = i'_{n+1}$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Byzantine}, \pi)$. By hypothesis, the number of Byzantine processes in π's first echo sample is identical in σ and σ': with a minor abuse of notation we effectively have $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{State})$. By hypothesis, $n+1 < \left|\eta(\alpha,\sigma)\right| = \left|\eta(\alpha,\sigma')\right|$, and we immediately get $r_{n+1} = r'_{n+1} = \emptyset$.

Since α is a sample-blind adversary, we have $i_{n+1} \neq (\texttt{Sample}, \pi, m)$.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \bot = r'_{n+1}$. By induction, we have that, for every $n < \left|\eta(\alpha,\sigma)\right|$, $\tau(\alpha,\sigma)_n = \tau(\alpha,\delta)_n$. □

Lemma 15. *Let* α *be a state-polling adversary, let* σ, σ' *be systems such that*

$$\eta(\alpha,\sigma) = \eta(\alpha,\sigma')$$

and, for all $\pi \in \Pi_C$, $m \in \mathcal{M}$,

$$|\{n \in 1..E \mid \sigma[\pi][m][n] \in \Pi_C\}| = \left|\{n \in 1..E \mid \sigma'[\pi][m][n] \in \Pi_C\}\right|$$

We have

$$\tau(\alpha,\sigma) = \tau(\alpha,\sigma')$$

Proof. The proof is similar to the proof of Lemma 14, and we omit it for the sake of brevity. The lemma is proved by induction and noting that, since α is a single-response adversary, it never invokes $State()$ throughout the second phase of an adversarial execution. □

In Section 3.11.8, we formally prove that the set of two-phase adversaries $\mathscr{A}_{tp}$ is optimal.

Before moving on to computing a bound on the adversarial power of $\mathscr{A}_{tp}$, we prove two additional lemmas on the behavior of two-phase adversaries.

Lemma 16. *Let α be a two-phase adversary. Let $\eta^{(i)}(\alpha,\sigma)$ denote the sequence of invocations in $\eta(\alpha,\sigma)$. Let σ,σ' be systems such that, for all $\pi \in \Pi_C$,*

$$\left|\sigma.Byzantine(\pi)\right| = \left|\sigma'.Byzantine(\pi)\right|$$

We have

$$\forall n \leq \min(\left|\eta(\alpha,\sigma)\right|, \left|\eta(\alpha,\sigma')\right|),\ \eta^{(i)}(\alpha,\sigma)_n = \eta^{(i)}(\alpha,\sigma')_n$$

Proof. The proof is similar to the proof of Lemma 14, and we omit it for the sake of brevity. The lemma is proved by induction and noting that, except for the last one, every response to (`State`) in $\eta(\alpha,\sigma)$, $\eta(\alpha,\sigma')$ is, by definition, $\emptyset$. □

Lemma 17. *Let α be a two-phase adversary. Let σ,σ' be systems such that $\left|\eta(\alpha,\sigma) = \left|\eta(\alpha,\sigma')\right|\right|$. Let $\theta^{(i)}(\alpha,\sigma)$ denote the sequence of invocations in $\theta(\alpha,\sigma)$ and, for all $\pi \in \Pi_C$,*

$$\left|\sigma.Byzantine(\pi)\right| = \left|\sigma'.Byzantine(\pi)\right|$$

We have

$$\theta^{(i)}(\alpha,\sigma) = \theta^{(i)}(\alpha,\sigma')$$

Proof. The proof is again similar to the proof of Lemma 14, and we omit it for the sake of brevity. The lemma is proved by induction and noting that:

- Since α is two-phase, it only invokes $State() \neq \emptyset$, the content of the $\left|\eta(\alpha,\sigma)\right|$-th response does not affect its behavior.
- Since α is single-response, it never invokes $State()$ throughout the second phase.

□

3.10 Consistency

In this section, we finally achieve the main goal of this chapter, i.e., to compute a bound on the ϵ-consistency of Sieve. In order to achieve this, in Section 3.6, we introduced Simplified Sieve, a strawman algorithm designed to be analytically tractable.

In Section 3.8, we proved that the consistency of Simplified Sieve is weaker than the consistency of Sieve. More precisely, we proved that an optimal adversary has a greater probability of compromising the consistency of Simplified Sieve than that of Sieve.

In doing so, we reduced the problem of bounding the ϵ-consistency of Sieve to that of bounding the adversarial power of a set of adversaries for Simplified Sieve that provably includes an optimal adversary.

Throughout Section 3.9, we employed a sequence of decorators to iteratively reduce the size of the set that provably includes an optimal adversary. Specifically, we proved that the set $\mathscr{A}_{tp}$ of two-phase adversaries is optimal. Intuitively, we proved that the behavior of an optimal adversary reduces to:

- (Echo phase): Causing every Byzantine process to send an `Echo`(m, m) message to every correct process, for every message m.
- (First phase): In sequence, causing correct processes to deliver a predefined sequence of messages until at least one correct process delivers a message.
- (Second phase): In sequence, causing the remaining set of correct process to deliver a predefined sequence of messages, determined only by the number of correct processes that pb.Delivered a message throughout the first phase.

In particular, the only information that we did not prove to be unnecessary to the Byzantine adversary is:

- The number of Byzantine processes in the first echo sample of each correct process π. This information is available to the adversary from the beginning of the adversarial execution, and does not change throughout the execution. We conjecture this information to still be of no use to the adversary, but we don't rely on this conjecture in proving what follows.
- The number of correct processes that pb.Deliver a message throughout the first phase of the adversarial execution, i.e., before at least one correct process delivers a message.

In this section, we redefine a two-phase adversary as a table of messages. In doing so, we provide a sound structure to a set of adversaries that provably includes an optimal one. We

then use this structure to analitically bound the probability of any two-phase adversary compromising the consistency of a random Simplified Sieve system.

First, we focus on the second phase of an adversarial execution, and study the probability of any two-phase adversary compromising the consistency of Simplified Sieve, given the number of correct processes that pb.Delivered, throughout the first phase, the message that was delivered by at least one correct process at the end of the first phase.

We then focus on the first phase of an adversarial execution, and study the probability of any two-phase adversary concluding the first phase of an adversarial execution having caused less than n correct processes to pb.Deliver m, m being the message that at least one correct process delivers at the end of the first phase.

We finally join the two above results to compute a bound ϵ_c on the probability of a two-phase adversary compromising the consistency of Simplified Sieve. Since at least one two-phase adversary is provably optimal, Simplified Sieve satisfies ϵ_c-consistency. Since the ϵ-consistency of Sieve is provably bound by the ϵ-consistency of Simplified Sieve, Sieve satisfies ϵ_c-consistency.

3.10.1 Two-phase adversaries

In Section 3.9, we proved that the set $\mathscr{A}_{tp}$ is optimal. In this section, we use Lemmas 16 and 17 to re-define the set of two-phase adversaries as a set of *triangular message tables*.

Definition 25 (Byzantine population)**.** A **Byzantine population** is a vector in the set

$$\mathscr{F} = (0..E)^{\Pi_C}$$

Let σ be a system. We define the **Byzantine population** of σ by

$$\forall \pi,\ F(\sigma)_\pi = \left|\sigma.Byzantine(\pi)\right|$$

Definition 26 (Two-phase adversary)**.** A **two-phase adversary** $\alpha \in \mathscr{A}_{tp}$ is a *triangular table* defined by:

$$\begin{aligned} &\alpha[F]_i \in \mathscr{M} && F \in \mathscr{F},\ i \in 1..C \\ &\alpha[F]_i^n \in \mathscr{M} && F \in \mathscr{F},\ n \in 0..C,\ i \in 1..(C-n) \end{aligned}$$

Coupled with a system σ, a two-phase adversary α:

- (Echo phase) Causes every Byzantine process to send an `Echo`(m, m) message to every correct process in σ, for every message m.
- (First phase) Sequentially causes $\zeta(1)$ to pb.Deliver $\alpha[F(\sigma)]_1$, $\zeta(2)$ to pb.Deliver $\alpha[F(\sigma)]_2$,

... in σ, until, as a result of the n-th pb.Delivery, at least one correct process delivers a message in σ. We note that, if σ is poisoned, then at least one correct process delivers a mesage in σ as a result of the echo phase, and $n = 0$.

- (Second phase) Sequentially causes $\zeta(n+1)$ to pb.Deliver $\alpha[F(\sigma)]^n_1, \ldots, \zeta(C)$ to pb.Deliver $\alpha[F(\sigma)]^n_{C-n}$ in σ.

3.10.2 Random variables

Let α be a two-phase adversary. In the next sections, we compute a bound on the probability of α compromising the consistency of a random, non-poisoned system. To this end, in this section we introduce a set of random variables.

Notation 8 (Delivery indicator)**.** Let σ be a system, let m, $m_1, \ldots, m_n$ be messages. We use

$$\delta_m[m_1, \ldots, m_n](\sigma) \in \{\texttt{True}, \texttt{False}\}$$

to indicate whether or not at least one correct process delivers m in σ, if $\zeta(1)$ pb.Delivers m_1, ..., $\zeta(n)$ pb.Delivers m_n in σ. We additionally define

$$\delta[m_1, \ldots, m_n](\sigma) = \bigvee_{m \in \mathcal{M}} \delta[m_1, \ldots, m_n](\sigma)$$

Let σ be a random, non-poisoned system. We define:

- **Byzantine population** $F_{\pi \in \Pi_C}(\sigma)$: represents the number of Byzantine processes in the first echo sample of π in σ.

- **First phase duration** $\eta(\sigma)$: represents the number of correct processes that pb.Deliver a message in the first phase, when α is coupled with σ. More formally,

$$\eta(\sigma) = \min n \mid (\delta[\alpha[F(\sigma)]_1, \ldots, \alpha[F(\sigma)]_n] = \texttt{True} \vee n = C)$$

- **First-phase deliveries** $S_{m \in \mathcal{M}}(\sigma)$: represents the number of correct processes that pb.Deliver message m throughout the first phase, when α is coupled with σ. More formally,

$$S_m(\sigma) = \left|\{n \in 1..\eta(\sigma) \mid \alpha[F(\sigma)]_n = m\}\right|$$

- **Second-phase deliveries** $T_{m \in \mathcal{M}}(\sigma)$: represents the number of correct processes that pb.Deliver message m throughout the second phase, when α is coupled with σ. More formally,

$$T_m(\sigma) = \left|\left\{n \in 1..(C - \eta(\sigma)) \mid \alpha[F(\sigma)]^{\eta(\sigma)}_n = m\right\}\right|$$

- **Deliveries** $N_{m \in \mathcal{M}}(\sigma)$: represents the number of correct processes that pb.Deliver mes-

sage m, when α is coupled with σ. More formally,

$$N_m(\sigma) = S_m(\sigma) + T_m(\sigma)$$

- **First delivered message** $H(\sigma) \in \mathcal{M} \cup \{\bot\}$: if, when α is coupled with σ, at least one correct process delivers a message, $H(\sigma)$ represents the first message to be delivered by at least one correct process in σ. Otherwise, $H(\sigma) = \bot$. More formally,

$$H(\sigma) = \begin{cases} \alpha[F(\sigma)]_{\eta(\sigma)} & \text{iff } \delta[\alpha[F(\sigma)]_1, \ldots, \alpha[F(\sigma)]_C] = \texttt{True} \\ \bot & \text{otherwise} \end{cases}$$

- **Correct echoes** $E^{k \in 0..C}_{m \in \mathcal{M}}[\pi](\sigma) \in 0..E \cup \{\bot\}$: if $k \leq N_i(\sigma)$, then $E^k_i[\pi](\sigma)$ represents the number of correct processes in π's echo sample for m that sent an $\texttt{Echo}(m, m)$ message to π in σ, when exactly k correct processes pb.Delivered m in σ. Otherwise, $E^k_m[\pi](\sigma) = \bot$.

- **Delivery** $A^{k \in 0..C}_{m \in \mathcal{M}}[\pi \in \Pi_C](\sigma) \in \{\texttt{True}, \texttt{False}, \bot\}$: if $k \leq N_m(\sigma)$, $A^k_m[\pi]$ represents, when α is coupled with σ, whether or not π delivered m after k correct processes pb.Delivered m. More formally,

$$A^k_m[\pi](\sigma) = \begin{cases} E^k_m[\pi] \geq \hat{E} - F_\pi & \text{iff } k \leq N_m(\sigma) \\ \bot & \text{otherwise} \end{cases}$$

- **Global delivery** $A^{k \in 0..C}_{m \in \mathcal{M}}(\sigma)$: if $k \leq N_m(\sigma)$, A^k_i represents, when α is coupled with σ, whether or not at least one process delivered m after k correct processes pb.Delivered m. More formally,

$$A^k_m(\sigma) = \begin{cases} \bigvee_{\pi \in \Pi_C} A^k_m[\pi](\sigma) & \text{iff } k \leq N_m(\sigma) \\ \bot & \text{otherwise} \end{cases}$$

- **First phase plan** $L_m(\sigma)$: represents the number of times m appears in the sequence

$$\alpha[F(\sigma)]_1, \ldots, \alpha[F(\sigma)]_C$$

  Intuitively, L_m represents the number of correct processes that α would eventually cause to pb.Deliver m, if no correct process ever delivered any message.

- **Adversarial success** W: W represents whether or not the adversary successfully compromises the consistency of the system.

We additionally define:

$$\begin{aligned}
E_m[\pi](\sigma) &= E_m^{N_m(\sigma)}[\pi](\sigma) \\
E_m^{(s)}[\pi](\sigma) &= E_m^{S_m(\sigma)}[\pi](\sigma) \\
E_m^{(t)}[\pi](\sigma) &= E_m[\pi](\sigma) - E_m^{(s)}[\pi](\sigma) \\
A_m[\pi](\sigma) &= A_m^{N_m(\sigma)}[\pi](\sigma) \\
A_m(\sigma) &= A^{N_m(\sigma)}(\sigma)
\end{aligned}$$

3.10.3 Byzantine population, correct echoes, delivery

In this section, we compute the probability distributions underlying Byzantine population. Given the Byzantine population, we then compute the number of correct echoes and the probability of delivery.

Byzantine population. As we discussed in Section 3.6.3, every correct process selects its first echo sample using the $Sample(\ldots)$ procedure, which, in turn, picks each element independently from the set of processes. Therefore, the number of correct processes in the first echo sample of each correct process is independently binomially distributed:

$$\mathscr{P}[\bar{F}_\pi] = \text{Bin}[E, f](\bar{F}_\pi)$$

Correct echoes. Let π be a correct process, let m be a message. If π has $\bar{F}_\pi$ Byzantine processes in its first echo sample and exactly k correct processes pb.Delivered m, then each of the $E - \bar{F}_\pi$ correct process in π's echo sample for m has an independent probability k/C of having pb.Delivered m.

Consequently, we have

$$\mathscr{P}\left[\bar{E}_m^k[\pi] \mid \bar{F}_\pi\right] = \begin{cases} \text{Bin}\left[E - \bar{F}_\pi, \frac{k}{C}\right]\left(\bar{E}_m^k[\pi]\right)\mathscr{P}\left[k \le N_m \mid \bar{F}_\pi\right] & \text{iff } \bar{E}_m^k[\pi] \ne \bot \\ \mathscr{P}\left[k > N_m \mid \bar{F}_\pi\right] & \text{otherwise} \end{cases}$$

We underline that the above holds true only because the adversary α is non-redundant. Indeed, since α knows the first phase duration η, it also knows H (this immediately follows from $H = \alpha[F]_\eta$). Therefore, if α was not non-redundant, the value of $E_m^k[\pi]$ would not necessarily be independent from the event $k \le N_m$.

We can see this with an example. With a minor slip of notation, consider an adversary α such

that

$$\begin{aligned} \alpha[\{0\}^{\Pi_C}]_1 &= 1 \\ \alpha[\{0\}^{\Pi_C}]_{i>1} &\neq 1 \\ \alpha[\{0\}^{\Pi_C}]_1^1 &= 1 \end{aligned}$$

We can immediately see that α is not non-redundant: if no correct process has any Byzantine process in its echo samples, and at least one correct process delivers 1 as an immediate result of $\zeta(1)$ pb.Delivering 1, α causes $\zeta(2)$ to pb.Deliver 1 again. If $\eta = 1$, then $l \geq 1$ correct process $\pi_1^*, \ldots, \pi_l^*$ exists such that $\zeta(1)$ appears at least $\hat{E}$ times in π_i^*'s echo sample for 1. Since a correct process π has a probability l/C of being among $\pi_1^*, \ldots, \pi_l^*$, if $N_1 \geq 2$ the distribution of $\bar{E}_m^k[\pi]$ becomes

$$\begin{aligned} &\mathscr{P}\left[\bar{E}_m^k[\pi] \mid F[\pi] = 0, N_1 \geq 2\right] \\ = \quad &\text{Bin}\left[E, \frac{k}{C}\right]\left(\bar{E}_m^k[\pi]\right)\left(\frac{C-l}{C} + \frac{l}{C}\frac{I(\bar{E}_m^k[\pi] \geq \hat{E})}{\sum_{e=\hat{E}}^{E} \text{Bin}\left[E, \frac{k}{C}\right](e)}\right) \end{aligned}$$

which is clearly not a binomial. Intuitively, if α was not non-redundant, the value of N_m may come to depend on whether or not m was delivered by at least one correct process, which obviously correlates with the value of $E_m^k[\pi]$.

Since α is non-redundant, however, and every correct process picks each echo sample independently, the value of $E_m^k[\pi]$ is indeed independent from the event $k \leq N_m$.

Delivery. Noting that a correct process π delivers a message m if it collects at least $\hat{E}$ `Echo(`m, m`)` messages from its echo sample for m, we can use the distribution underlying the correct echoes to obtain

$$\mathscr{P}\left[A_m^k[\pi] \mid \bar{F}_\pi\right] = \sum_{\bar{E}_m^k[\pi] = \hat{E} - \bar{F}_\pi}^{E - \bar{F}_\pi} \mathscr{P}\left[\bar{E}_m^k[\pi] \mid \bar{F}_\pi\right]$$

and, using the law of total probability, we get

$$\mathscr{P}\left[A_m^k[\pi]\right] = \sum_{\bar{F}_\pi = 0}^{E} \mathscr{P}\left[A_m^k[\pi] \mid \bar{F}_\pi\right] \mathscr{P}\left[\bar{F}_\pi\right]$$

Finally, since the above holds independently for every process π, we have

$$\mathscr{P}\left[A_m^k\right] = 1 - \prod_{\pi \in \Pi_C}\left(1 - \mathscr{P}\left[A_m^k[\pi]\right]\right) = 1 - \left(1 - \mathscr{P}\left[A_m^k[\zeta(1)]\right]\right)^C$$

3.10.4 Second phase

In the previous sections, we computed the probability of any correct process delivering a message m, given that k correct processes pb.Delivered m. In Section 3.10.1, we discussed how an optimal adversarial execution unfolds in two phases: the first takes place before any correct process delivers any message; throughout the second, the goal of the adversary is to cause at least one correct process to deliver one additional message.

In this section, we focus on the second phase. We assume that a message H has already been delivered by at least one correct process. Given the number of correct processes that pb.Delivered each message throughout the first phase, we compute (where possible) a bound on the probability of any message different from H being delivered before the end of the adversarial execution, i.e., the probability of the adversary successfully compromising the consistency of the system.

Correct echoes for a non-delivered message. Let π be a correct process that has $\bar{F}$ Byzantine processes in its first echo sample. Let m be a message such that π does not deliver m after k correct processes pb.Delivered m. Here we use Bayes' theorem to compute the probability distribution underlying the number of correct echoes received by π for m.

Notation 9 (Indicator function)**.** We use I to denote the **indicator function**. Let c be a predicate, then

$$I(c) = \begin{cases} 1 & \text{iff } c \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathscr{P}\left[\bar{E}_m^k[\pi] \mid \cancel{A_m^k[\pi]}, \bar{F}_\pi\right] &= \mathscr{P}\left[\bar{E}_m^k[\pi] \mid E_m^k[\pi] < \hat{E} - \bar{F}_\pi, \bar{F}_\pi\right] \\
&= \frac{\mathscr{P}\left[E_m^k[\pi] < \hat{E} - \bar{F}_\pi \mid \bar{E}_m^k[\pi], \bar{F}_\pi\right] \mathscr{P}\left[\bar{E}_m^k[\pi] \mid \bar{F}_\pi\right]}{\mathscr{P}\left[E_m^k[\pi] < \hat{E} - \bar{F}_\pi \mid \bar{F}_\pi\right]} \\
&= \frac{I(\bar{E}_m^k[\pi] < \hat{E} - \bar{F}_\pi) \mathscr{P}\left[\bar{E}_m^k[\pi] \mid \bar{F}_\pi\right]}{\sum_{e=0}^{\hat{E}-\bar{F}_\pi-1} \mathscr{P}\left[E_m^k[\pi] = e \mid \bar{F}_\pi\right]}
\end{aligned}$$

where the numerator of the last term includes an indicator function because any condition $A < B$, given $\bar{A}$ and $\bar{B}$, is always satisfied deterministically.

Conditions. Let π be a correct process, let m be a message. Throughout the rest of this section, we compute the probability of π eventually delivering m under the following conditions:

- $\bar{F}_\pi$ processes in π's first echo sample are Byzantine.
- m is not the message that is delivered at the end of the first phase, i.e., $H \neq m$.

- $\bar{S}_m$ correct processes pb.Deliver m throughout the first phase.
- $\bar{T}_m$ correct processes pb.Deliver m throughout the second phase.

First phase correct echoes. Here we compute, under the above conditions, the probability distribution underlying $E_m^{(s)}[\pi]$, i.e., the number of correct echoes that π collects for m throughout the first phase.

Since $H \neq m$, π does not deliver m throughout the first phase. In other words, π does not deliver m after $\bar{S}_m$ correct processes pb.Delivered m, and we immediately have

$$\mathscr{P}\left[\bar{E}_m^{(s)}[\pi] \mid H \neq m, \bar{S}_m, \bar{F}_\pi\right] = \mathscr{P}\left[E_m^{\bar{S}_m}[\pi] = \bar{E}_m^{(s)}[\pi] \mid \cancel{A_m^{\bar{S}_m}[\pi]}, \bar{F}_\pi\right]$$

Second phase correct echoes. Here we compute, under the above conditions and given $\bar{E}_m^{(s)}[\pi]$, the probability distribution underlying $\bar{E}_m^{(t)}[\pi]$, i.e., the number of correct echoes that π collects for m throughout the second phase.

We start by noting that, out of the E elements in π's echo sample for m:

- $\bar{F}_\pi$ are Byzantine.
- $\bar{E}_m^{(s)}[\pi]$ belong to the set of $\bar{S}_m$ processes that pb.Delivered m throughout the first phase.
- $E - \bar{F}_\pi - \bar{E}_m^{(s)}[\pi]$ belong to the set of $C - \bar{S}_m$ processes that did not pb.Deliver m throughout the first phase.

Moreover, out of the $C - \bar{S}_m$ processes that did not pb.Deliver m throughout the first phase, $\bar{T}_m$ pb.Delivered m throughout the second phase. Therefore, each of the processes in π's echo sample for m that did not pb.Deliver m throughout the first phase has an independent probability $\bar{T}_m / (C - \bar{S}_m)$ of pb.Delivering m throughout the second phase.

Consequently, $\bar{E}_m^{(t)}$ is binomially distributed:

$$\mathscr{P}\left[\bar{E}_m^{(t)} \mid \bar{E}_m^{(s)}, \bar{S}_m, \bar{T}_m, \bar{F}_\pi\right] = \text{Bin}\left[E - \bar{F}_\pi - \bar{E}_m^{(s)}, \frac{\bar{T}_m}{C - \bar{S}_m}\right]\left(\bar{E}_m^{(t)}[\pi]\right)$$

Delivery probability (given message). We can finally compute, under the above conditions, the probability of π eventually delivering m.

We start by expanding the definition of $A_m[\pi]$ to get

$$\begin{aligned} &\mathscr{P}\left[A_m[\pi] \mid H \neq m, \bar{S}_m, \bar{T}_m, \bar{F}_\pi\right] \\ = \quad &\mathscr{P}\left[E_m[\pi] \geq \hat{E} - \bar{F}_\pi \mid H \neq m, \bar{S}_m, \bar{T}_m, \bar{F}_\pi\right] = (\star_1) \end{aligned}$$

and then expand the definition of $E_m[\pi]$ to get

$$(\star_1) = \mathscr{P}\left[E_m^{(t)}[\pi] \geq \hat{E} - \bar{F}_\pi - E_m^{(s)}[\pi] \mid H \neq m, \bar{S}_m, \bar{T}_m, \bar{F}_\pi\right] = (\star_2)$$

Finally, using the law of total probability on each possible value of $E_m^{(s)}[\pi]$, we get

$$\begin{aligned}(\star_2) \quad = \quad \sum_{\bar{E}_m^{(s)}[\pi]=0}^{E-\bar{F}_\pi} \Bigg(& \underbrace{\mathscr{P}\left[E_m^{(t)}[\pi] \geq \hat{E} - \bar{F}_\pi - \bar{E}_m^{(s)}[\pi] \mid \bar{E}_m^{(s)}[\pi], \bar{S}_m, \bar{T}_m, \bar{F}_\pi\right]}_{(\star_a)} \\ & \cdot \underbrace{\mathscr{P}\left[\bar{E}_m^{(s)}[\pi] \mid H \neq m, \bar{S}_m, \bar{F}_\pi\right]}_{(\star_b)} \Bigg)\end{aligned}$$

As we previously established,

$$\mathscr{P}\left[\bar{E}_m^{(t)}[\pi] \mid E_m^{(s)}[\pi] = i, \bar{S}_m, \bar{T}_m, \bar{F}_\pi\right] = \mathscr{P}\left[X_i = \bar{E}_m^{(t)}[\pi]\right]$$

with

$$\begin{aligned} X_i &\sim \mathrm{Bin}\left[A - i, p\right] \\ A &= E - \bar{F}_\pi \\ p &= \frac{\bar{T}_m}{(C - \bar{S}_m)} \end{aligned}$$

Moreover,

$$(\star_a) = \mathscr{P}[X_i \geq B - i]$$

with

$$B = \hat{E} - \bar{F}_\pi \leq E - \bar{F}_\pi = A$$

Therefore, following from Lemma 7, $(\star_a)$ is an increasing function of $\bar{E}_m^{(s)}[\pi]$. Moreover, as we previously established,

$$\begin{aligned} \mathscr{P}\left[\bar{E}_m^{(s)}[\pi] \mid H \neq m, \bar{S}_m, \bar{F}_\pi\right] &= \mathscr{P}\left[E_m^{\bar{S}_m}[\pi] = \bar{E}_m^{(s)}[\pi] \mid \cancel{A_m^{\bar{S}_m}[\pi]}, \bar{F}_\pi\right] \\ &= \frac{I(\bar{E}_m^{\bar{S}_m}[\pi] < \hat{E} - \bar{F}_\pi)\mathscr{P}\left[\bar{E}_m^{\bar{S}_m}[\pi] \mid \bar{F}_\pi\right]}{\sum_{e=0}^{\hat{E}-\bar{F}_\pi-1} \mathscr{P}\left[E_m^{\bar{S}_m}[\pi] = e \mid \bar{F}_\pi\right]} \end{aligned}$$

and $(\star_2)$ can be restated as

$$\begin{aligned}
(\star_2) &= \frac{\sum_{x=0}^{K-l} f(x) g(x)}{\sum_{x=0}^{K-l} g(x)} \\
K &= E - \bar{F}_\pi \\
l &= E - \hat{E} + 1 \\
f(x) &= \mathscr{P}\left[E_m^{(t)}[\pi] \geq \hat{E} - \bar{F}_\pi - x \mid E_m^{(s)}[\pi] = x, \bar{S}_m, \bar{T}_m, \bar{F}_\pi\right] \\
g(x) &= \mathscr{P}\left[E_m^{\bar{S}_m}[\pi] = x \mid \bar{F}_\pi\right]
\end{aligned}$$

with $f(x)$ increasing and $\sum_{x=0}^{K} g(x) = 1$. Following from Corollary 1, we therefore have

$$\begin{aligned}
&\mathscr{P}\left[A_m[\pi] \mid H \neq m, \bar{S}_m, \bar{T}_m, \bar{F}_\pi\right] \\
\leq & \sum_{\bar{E}_m^{(s)}[\pi]=0}^{E-\bar{F}_\pi} \mathscr{P}\left[\bar{E}_m^{(s)}[\pi] + E_m^{(t)}[\pi] \geq \hat{E} - \bar{F}_\pi \mid \bar{E}_m^{(s)}[\pi], \bar{S}_m, \bar{T}_m, \bar{F}_\pi\right] \\
& \qquad \cdot \mathscr{P}\left[E_m^{\bar{S}_m}[\pi] = \bar{E}_m^{(s)}[\pi] \mid \bar{F}_\pi\right] \\
= & \; (\star_3)
\end{aligned}$$

which, as we previously established, can be restated as

$$\begin{aligned}
(\star_3) &= \mathscr{P}[X + Y \geq H] = \sum_{K=H}^{A} \mathscr{P}[X + Y = K] \\
\mathscr{P}\left[\bar{X}\right] &= \text{Bin}\left[A, \frac{x}{B}\right]\left(\bar{X}\right) \\
\mathscr{P}\left[\bar{Y} \mid \bar{X}\right] &= \text{Bin}\left[A - \bar{X}, \frac{y}{B - x}\right]\left(\bar{Y}\right) \\
H &= \hat{E} - \bar{F}_\pi \\
A &= E - \bar{F}_\pi \\
B &= C \\
x &= \bar{S}_m \\
y &= \bar{T}_m
\end{aligned}$$

which, using Lemma 6, yields the bound

$$\mathscr{P}\left[A_m[\pi] \mid H \neq m, \bar{S}_m, \bar{T}_m, \bar{F}_\pi\right] \leq \sum_{e=\hat{E}-\bar{F}_\pi} \text{Bin}\left[E - \bar{F}_\pi, \frac{\bar{S}_m + \bar{T}_m}{C}\right](e) \tag{3.2}$$

Delivery probability (any message). We now move on to compute the probability that a correct process π will eventually deliver any message other than H, under the following assumptions:

- The first phase of the adversarial execution is concluded.
- The number $\bar{F}_\pi$ of Byzantine processes in the first echo sample of π is given.
- The number $\bar{S}_m$, $\bar{T}_m$ of correct processes that pb.Delivered each message m throughout the first and second phase respectively is given.

Since every echo sample is picked independently, from Equation (3.2) follows

$$\begin{aligned}
\mathscr{P}\left[\bigvee_{m \neq \bar{H}} A_m[\pi] \mid \bar{S}_1, \ldots, \bar{S}_C, \bar{T}_1, \ldots, \bar{T}_C, \bar{F}_\pi\right] &\leq \mathscr{P}\left[\bigvee_{i \neq m} (X_i \geq K)\right] \\
\mathscr{P}[\bar{X}_i] &= \text{Bin}[N, p_i](\bar{X}_i) \\
N &= E - \bar{F}_\pi \\
K &= \hat{E} - \bar{F}_\pi \\
p_i &= \frac{\bar{S}_i + \bar{T}_i}{C}
\end{aligned}$$

and noting that

$$\sum_{m \neq \bar{H}} \frac{\bar{S}_m + \bar{T}_m}{C} = \sum_{n \neq \bar{H}} \bar{N}_m = C - \bar{N}_{\bar{H}}$$

we can use Lemma 10 to obtain the bound

$$\mathscr{P}\left[\bigvee_{m \neq \bar{H}} A_m[\pi] \mid \bar{S}_1, \ldots, \bar{S}_C, \bar{T}_1, \ldots, \bar{T}_C, \bar{F}_\pi\right] \leq \phi(\bar{N}_{\bar{H}}, \bar{F}_\pi)$$

with

$$\phi(\bar{N}_{\bar{H}}, \bar{F}_\pi) = \begin{cases} \alpha(\bar{N}_{\bar{H}}, \bar{F}_\pi) \cdot \beta(\bar{N}_{\bar{H}}, \bar{F}_\pi) & \text{iff } \frac{C - \bar{N}_{\bar{H}}}{C} \leq \frac{(\hat{E} - \bar{F}_\pi) - \sqrt{\hat{E} - \bar{F}_\pi}}{E - \bar{F}_\pi} \\ 1 & \text{otherwise} \end{cases} \tag{3.3}$$

where

$$\alpha(\bar{N}_{\bar{H}}, \bar{F}_\pi) = \left(\frac{e(E - \bar{F}_\pi)\frac{C - \bar{N}_{\bar{H}}}{C}}{\hat{E} - \bar{F}_\pi}\right)^{(\hat{E} - \bar{F}_\pi)}$$

$$\beta(\bar{N}_{\bar{H}}, \bar{F}_\pi) = \exp\left(-(E - \bar{F}_\pi)\frac{C - \bar{N}_{\bar{H}}}{C}\right)$$

At a first glance, the second branch of the bound above could seem unreasonably lax. We underline, however, that for a large enough $(\hat{E} - \bar{F}_\pi)$,

$$\frac{(\hat{E} - \bar{F}_\pi) - \sqrt{\hat{E} - \bar{F}_\pi}}{E - \bar{F}_\pi} \simeq \frac{\hat{E} - \bar{F}_\pi}{E - \bar{F}_\pi}$$

and, since the median of $\text{Bin}[N, p]$ is either $\lceil Np \rceil$ or $\lfloor Np \rfloor$,

$$\sum_{e=\hat{E}-\bar{F}_\pi}^{E-\bar{F}_\pi} \text{Bin}\left[E-\bar{F}_\pi, \frac{\hat{E}-\bar{F}_\pi}{E-\bar{F}_\pi}\right](e) \simeq \frac{1}{2}$$

Therefore, even in the second branch, the bound introduces a limited multiplicative error. Moreover, as we will see in the numerical analysis, the error introduced by the bound is non-negligible only for extremely unlikely values of $\bar{N}_{\bar{H}}$.

Adversarial success probability. Throughout this section, we computed the probability that a correct process π will deliver a message different from the message that was delivered throughout the first phase.

We showed that such probability can be bound by a function that only depends on the number of Byzantine processes in the first echo sample of π, and the number of correct processes that pb.Delivered H throughout the first phase.

We therefore have

$$\mathscr{P}\left[\bigvee_{m\neq H} A_m[\pi] \mid \bar{N}_H, \bar{F}_\pi\right] \leq \phi(\bar{N}_H, \bar{F}_\pi)$$

By the law of total probability we have

$$\begin{aligned}
\mathscr{P}\left[\bigvee_{m\neq H} A_m[\pi] \mid \bar{N}_H\right] &= \sum_{m\neq H} \mathscr{P}\left[\bigvee_{m\neq H} A_m[\pi] \mid \bar{N}_H, \bar{F}_\pi\right] \mathscr{P}\left[\bar{F}_\pi \mid \bar{N}_H\right] \\
&\leq \sum_{m\neq \bar{H}} \phi(\bar{N}_H, \bar{F}_\pi) \mathscr{P}\left[\bar{F}_\pi \mid \bar{N}_H\right]
\end{aligned}$$

Since H was delivered by at least one correct process at the end of the first phase, we know that:

- One correct process π^+ delivered H immediately after $\bar{N}_H$ correct processes pb.Delivered H.

- Every other correct process did not deliver H before $\bar{N}_H$ correct processes pb.Delivered H.

We start by computing the probability distribution underlying $\bar{F}_{\pi^+}$. Using Bayes' theorem we

get

$$
\begin{aligned}
\mathscr{P}\left[\bar{F}_{\pi^+} \mid \bar{N}_H\right] &= \mathscr{P}\left[\bar{F}_{\pi^+} \mid A_H^{\bar{N}_H}[\pi^+], \cancel{A_H^{\bar{N}_H-1}[\pi^+]}\right] \\
&= \frac{\mathscr{P}\left[A_H^{\bar{N}_H}[\pi^+], \cancel{A_H^{\bar{N}_H-1}[\pi^+]} \mid \bar{F}_{\pi^+}\right]\mathscr{P}\left[\bar{F}_{\pi^+}\right]}{\mathscr{P}\left[A_H^{\bar{N}_H}[\pi^+], \cancel{A_H^{\bar{N}_H-1}[\pi^+]}\right]}
\end{aligned}
$$

and noting that $A_H^{\bar{N}_H-1}[\pi^+] \implies A_H^{\bar{N}_H}[\pi^+]$, we have

$$
\begin{aligned}
\mathscr{P}\left[A_H^{\bar{N}_H}[\pi^+], \cancel{A_H^{\bar{N}_H-1}[\pi^+]}\right] &= \mathscr{P}\left[A_H^{\bar{N}_H}[\pi^+]\right] - \mathscr{P}\left[A_H^{\bar{N}_H-1}[\pi^+]\right] \\
\mathscr{P}\left[A_H^{\bar{N}_H}[\pi^+], \cancel{A_H^{\bar{N}_H-1}[\pi^+]} \mid \bar{F}_{\pi^+}\right] &= \mathscr{P}\left[A_H^{\bar{N}_H}[\pi^+] \mid \bar{F}_{\pi^+}\right] \\
&\quad - \mathscr{P}\left[A_H^{\bar{N}_H-1}[\pi^+] \mid \bar{F}_{\pi^+}\right]
\end{aligned}
$$

Similarly, for $\pi^- \neq \pi^+$, we get

$$
\begin{aligned}
\mathscr{P}\left[\bar{F}_{\pi^-} \mid \bar{N}_H\right] &= \mathscr{P}\left[\bar{F}_{\pi^-} \mid \cancel{A_H^{\bar{N}_H-1}[\pi^-]}\right] \\
&= \frac{\mathscr{P}\left[\cancel{A_H^{\bar{N}_H-1}[\pi^-]} \mid \bar{F}_{\pi^-}\right]\mathscr{P}\left[\bar{F}_{\pi^-}\right]}{\mathscr{P}\left[\cancel{A_H^{\bar{N}_H-1}[\pi^-]}\right]}
\end{aligned}
$$

Since each correct process picks its echo sample independently, we have

$$
\begin{aligned}
\mathscr{P}\left[W \mid \bar{N}_H\right] &= 1 - \prod_{\pi \in \Pi_C}\left(1 - \mathscr{P}\left[\bigvee_{m \neq H} A_m[\pi] \mid \bar{N}_H\right]\right) \\
&\leq 1 - \left(1 - \phi^+(\bar{N}_H)\right)\left(1 - \phi^-(\bar{N}_H)\right)^{C-1}
\end{aligned} \tag{3.4}
$$

with

$$
\begin{aligned}
\phi^+(\bar{N}_H) &= \sum_{\bar{F}_{\pi^+}=0}^{E} \phi(\bar{N}_H, \bar{F}_{\pi^+})\mathscr{P}\left[\bar{F}_{\pi^+} \mid \bar{N}_H\right] \\
\phi^-(\bar{N}_H) &= \sum_{\bar{F}_{\pi^-}=0}^{E} \phi(\bar{N}_H, \bar{F}_{\pi^-})\mathscr{P}\left[\bar{F}_{\pi^-} \mid \bar{N}_H\right]
\end{aligned}
$$

3.10.5 First phase

In the previous section, we computed, given the number of correct processes that pb.Delivered the first delivered message, the probability of a two-phase adversary successfully compromising the consistency of a system.

In this section, we compute the probability distribution underlying the number of correct processes that pb.Deliver the first delivered message.

Definition 27 (Deafened adversary)**.** Let α be a two-phase adversary. We define $\Delta(\alpha)$ the **deafened version of** α if:

- $\Delta(\alpha)$ is a process-sequential adversary.
- Coupled with a system σ, $\Delta(\alpha)$ sequentially causes the pb.Delivery of $\alpha[F(\sigma)]_1, \ldots, \alpha[F(\sigma)]_C$.

Intuitively, the deafened version of a two-phase adversary α is an adversary whose adversarial execution would be identical to α's, if no correct process ever delivered any message.

Lemma 18. *Let α be a two-phase adversary, let σ be a system. We have*

$$\eta(\alpha, \sigma) = \eta(\Delta(\alpha), \sigma)$$

Proof. It follows immediately from Definition 27: $\Delta(\alpha)$ causes the same processes to pb.Deliver the same messages as α throughout the first phase. □

Definition 28 (Delivery cost)**.** Let α be an auto-echo adversary, let σ be a non-poisoned system, let m be a message such that, when α is coupled with σ, at least one correct process delivers m. We define the **delivery cost of** m $\lambda(\alpha, \sigma, m)$ as the minimum $\lambda \in 1..C$ such that, when α is coupled with σ, at least one correct process delivers m after λ correct processes pb.Delivered m.

Lemma 19. *Let α be a two-phase adversary, let σ be a non-poisoned system such that, when coupled with σ, α causes at least one correct process to deliver one message. Let $\bar{H}$ be the first message delivered by at least one correct process, when α is coupled with σ.*

We have that

$$\lambda(\alpha, \sigma, \bar{H}) \geq \min_{m \in \mathcal{M}} \lambda(\Delta(\alpha), \sigma, m)$$

Proof. Following from Lemma 18 $\eta(\alpha, \sigma) = \eta(\Delta(\alpha), \sigma)$. Therefore, at least one correct process delivers $\bar{H}$ after $\lambda(\alpha, \sigma, \bar{H})$ processes pb.Deliver $\bar{H}$, when $\Delta(\alpha)$ is coupled with σ. □

In this section, we bound the cumulative probability $\mathscr{P}[N_H \leq L]$ for an adversary α by bounding the probability that the deafened adversary $\Delta(\alpha)$ will cause the delivery of at least one message m, with a cost smaller or equal to L.

Let m be a message. We start by noting that, by definition, $\Delta(\alpha)$ eventually causes L_m correct processes to pb.Deliver m. Let π be a correct process, let $\bar{F}_\pi$ be the number of Byzantine processes in π's first echo sample.

We denote with $\Lambda_m[\pi]$ the random variable representing the minimum number of correct processes that pb.Deliver m, before π delivers m. If π never delivers m, we set $\Lambda_m[\pi] = \infty$.

Let $L \in 1..C$. Using the tools we developed in the previous section, we immediately get

$$\mathscr{P}\left[\Lambda_m[\pi] \le L \mid \bar{L}_m, \bar{F}_\pi\right] = \sum_{e=\hat{E}-\bar{F}_\pi}^{E-\bar{F}_\pi} \mathrm{Bin}\left[E - \bar{F}_\pi, \frac{\min\left(\bar{L}_m, L\right)}{C}\right](e)$$

and using the independence of echo samples, we get

$$\begin{aligned}
\mathscr{P}\left[\bigvee_{m\in\mathcal{M}} \Lambda_m[\pi] \le L \mid \bar{L}_1, \ldots, \bar{L}_C, \bar{F}_\pi\right] &= \mathscr{P}\left[\bigvee_{i\in\mathcal{M}} (X_i \ge K)\right] \\
\mathscr{P}\left[\bar{X}_i\right] &= \mathrm{Bin}\left[N, p_i\right]\left(\bar{X}_i\right) \\
N &= E - \bar{F}_\pi \\
K &= \hat{E} - \bar{F}_\pi \\
p_i &= \frac{\min\left(\bar{L}_i, L\right)}{C}
\end{aligned}$$

and we can use Lemma 10 to obtain the bound

$$\begin{aligned}
&\mathscr{P}\left[\bigvee_{m\in\mathcal{M}} \Lambda_m[\pi] \le L \mid \bar{L}_1, \ldots, \bar{L}_C, \bar{F}_\pi\right] \\
\le\ & 1 - \left(1 - \psi(L, \bar{F}_\pi)\right)^{\lfloor \frac{C}{L} \rfloor}\left(1 - \psi(C \bmod L, \bar{F}_\pi)\right)
\end{aligned}$$

with

$$\psi(M, \bar{F}_\pi) = \begin{cases} \alpha(M, \bar{F}_\pi) \cdot \beta(M, \bar{F}_\pi) & \text{iff } \frac{M}{C} \le \frac{(\hat{E}-\bar{F}_\pi) - \sqrt{\hat{E}-\bar{F}_\pi}}{E-\bar{F}_\pi} \\ 1 & \text{otherwise} \end{cases} \tag{3.5}$$

where

$$\alpha(M, \bar{F}_\pi) = \left(\frac{e(E - \bar{F}_\pi)\frac{M}{C}}{\hat{E} - \bar{F}_\pi}\right)^{(\hat{E}-\bar{F}_\pi)}$$

$$\beta(M, \bar{F}_\pi) = \exp\left(-(E - \bar{F}_\pi)\frac{M}{C}\right)$$

Noting that the bound holds for any value of $\bar{L}_1, \ldots, \bar{L}_C$, we can use again the law of total probability to obtain

$$\mathscr{P}\left[\bigvee_{m\in\mathcal{M}} \Lambda_m[\pi] \le L\right] \le \psi(L)$$

with

$$\psi(L) = \sum_{\bar{F}_\pi=0}^{E} 1 - \left(1 - \psi(L, \bar{F}_\pi)\right)^{\lfloor \frac{C}{L} \rfloor}\left(1 - \psi(C \bmod L, \bar{F}_\pi)\right)\mathscr{P}\left[\bar{F}_\pi\right] \tag{3.6}$$

and using the independence of echo samples across correct processes we finally get

$$\mathscr{P}\left[\bigvee_{\pi\in\Pi_C, m\in\mathscr{M}} \Lambda_m[\pi] \le L\right] \le 1-(1-\psi(L))^C \tag{3.7}$$

We now have all the elements to prove

Theorem 9. *Sieve satisfies ϵ_c-consistency, with*

$$\begin{aligned}
\epsilon_c &\le \epsilon_p + \sum_{L=0}^{C} \tilde{\psi}(L)\tilde{\phi}(L) \\
\tilde{\psi}(L) &= \begin{cases} \left(1-(1-\psi(L))^C\right) - \left(1-(1-\psi(L-1))^C\right) & \mathit{iff}\, L \in 1..C \\ 0 & \mathit{iff}\, L \in \{-1,0\} \\ 1 & \mathit{iff}\, L = C \end{cases} \\
\tilde{\phi}(L) &= \left(1-(1-\phi^+(L))(1-\phi^-(L))^{C-1}\right) \\
\epsilon_p &= 1-\left(1-\sum_{\bar{F}=\hat{E}}^{E} Bin[E,f](\bar{F})\right)^C
\end{aligned}$$

Proof. Following from Lemma 19, we have

$$\mathscr{P}[N_H \le L] \le \mathscr{P}\left[\bigvee_{\pi\in\Pi_C, m\in\mathscr{M}} \Lambda_m[\pi] \le L\right] \tag{3.8}$$

By the law of total probability, we have

$$\begin{aligned}
\mathscr{P}[W] &= \sum_{x=0}^{C} \left(f(x)\left(g(x)-g(x-1)\right)\right) \\
f(x) &= \mathscr{P}\left[W \mid \bar{N}_H = x\right] \\
g(x) &= \mathscr{P}[N_H \le x]
\end{aligned}$$

and from Lemma 9 we get

$$\begin{aligned}
\mathscr{P}[W] &\le \sum_{x=0}^{C} \left(f(x)(h(x)-h(x-1))\right) \\
h(x) &= \mathscr{P}\left[\bigvee_{\pi\in\Pi_C, m\in\mathscr{M}} \Lambda_m[\pi] \le x\right]
\end{aligned}$$

The probability of compromising a non-poisoned system is obtained by applying the bounds in Equations (3.4), (3.7) and (3.8).

It is easy to see that ϵ_p represents the probability of a random system being poisoned: indeed,

each correct process has an independent probability

$$\sum_{\bar{F}=\hat{E}}^{E} \text{Bin}[E, f](\bar{F})$$

of having more than $\hat{E}$ Byzantine processes in its first echo sample, i.e., of being poisoned.

Therefore, the bound on ϵ_c bounds the probability of any two-phase adversary compromising the consistency of a cob system. Due to Lemma 27, the set $\mathscr{A}_{tp}$ of two-phase adversaries is optimal. Therefore, Simplified Sieve satisfies ϵ_c-consistency.

Due to Lemma 12, the adversarial power of an optimal pcb adversary is bound by the adversarial power of an optimal cob adversary, and the theorem is proved. □

3.11 Decorators

In this section, we provide the proof that each of the sets of cob adversaries presented in Section 3.9 is optimal.

3.11.1 Auto-echo adversary

Lemma 20. *The set of auto-echo adversaries $\mathscr{A}_{ae}$ is optimal.*

Proof. We prove the result using a **decorator**, i.e., an algorithm that acts as an interface between an adversary and a system. An adversary coupled with a decorator effectively implements an adversary. Here we show that a decorator Δ_{ae} exists such that, for every $\alpha \in \mathscr{A}$, the adversary $\alpha' = \Delta_{ae}(\alpha)$ is an auto-echo adversary, and more powerful than α. If this is true, then the lemma is proved: let α^* be an optimal adversary, then the auto-echo adversary $\alpha^+ = \Delta_{ae}(\alpha^*)$ is optimal as well.

Decorator. Algorithm 7 implements **Auto-echo decorator**, a decorator that transforms an adversary into an auto-echo adversary. Provided with an adversary adv, Auto-echo decorator acts an interface between adv and a system sys, effectively implementing an auto-echo adversary $aeadv$. Auto-echo decorator exposes both the adversary and the system interfaces: the underlying adversary adv uses $aeadv$ as its system.

Auto-echo decorator works as follows:

- Procedure $aeadv.Init()$ initializes the following variables:
  - A $queue$ list that contains every combination of (π, m, ξ), π being a correct process, m being a message and ξ being a Byzantine process: $queue$ is used to initially

Algorithm 7 Auto-echo decorator.

```
1: Implements:
2:     AutoEchoAdversary + CobSystem, instance aeadv
3:
4: Uses:
5:     CobAdversary, instance adv, system aeadv
6:     CobSystem, instance sys
7:
8: procedure aeadv.Init() is
9:     queue = ∅;
10:
11:    for all π ∈ Π_C do
12:        for all m ∈ ℳ do
13:            for all ξ ∈ Π \ Π_C do
14:                queue ← queue ∪ {(π, m, ξ)};
15:            end for
16:        end for
17:    end for
18:
19:    echoes = {⊥}^(C×C×N);                    ▷ C × C × N table filled with ⊥.
20:    executed = False;
21:    adv.Init();
22:
23: procedure aeadv.Step() is
24:    if queue ≠ ∅ then
25:        (π, m, ξ) = queue[1];
26:        sys.Echo(π, m, ξ, m);
27:        queue ← queue \ {(ξ, π, m)};
28:    else
29:        executed ← False;
30:        while executed = False do
31:            adv.Step();
32:        end while
33:    end if
34:
35: procedure aeadv.Byzantine(process) is
36:    return sys.Byzantine(process);
37:
```

```
38: procedure aeadv.State() is
39:     state = ∅;
40:
41:     for all (π, m) ∈ sys.State() do
42:         n = 0;
43:
44:         for all ρ ∈ sys.Sample(π, m) do
45:             if echoes[π][m][ρ] = m then
46:                 n ← n + 1;
47:             end if
48:         end for
49:
50:         if n ≥ Ê then
51:             state ← state ∪ {(π, m)};
52:         end if
53:     end for
54:
55:     return state;
56:
57: procedure aeadv.Sample(process, message) is
58:     sample = ∅;
59:
60:     for all ρ ∈ sys.Sample(process, message) do
61:         if echoes[process][message][ρ] ≠ ⊥ then
62:             sample ← sample ∪ {ρ};
63:         end if
64:     end for
65:
66:     return sample;
67:
68: procedure aeadv.Deliver(process, message) is
69:     executed = True;
70:
71:     for all π ∈ Π_C do
72:         for all m ∈ ℳ do
73:             echoes[π][m][process] = message;
74:         end for
75:     end for
76:
77:     sys.Deliver(process, message, flag);
78:
79: procedure aeadv.Echo(process, sample, source, message) is
80:     echoes[process][sample][source] = message;
81:
```

```
82: procedure aeadv.End() is
83:     executed = True;
84:     sys.End();
85:
```

cause every Byzantine process ξ to send an `Echo`(m, m) message to every correct process π, for every message m.

- An *echoes* table, initialized with $\perp$ values: *echoes* is used to keep track of all the `Echo` messages that would have been sent to each correct process in *sys*, if *adv* was playing instead of *aeadv*.

- Procedure $aeadv.Step()$ checks if *queue* is not empty. If it is not empty, it pops (i.e., picks and removes) its first element (π, m, ξ), with $\xi \in \Pi \setminus \Pi_C$, $\pi \in \Pi_C$ and $m \in \mathscr{M}$. It then causes ξ to send π an `Echo`(m, m) message.

  If *queue* is empty instead, the procedure calls $adv.Step()$ until either $sys.Deliver(\ldots)$ or $sys.End()$ are called: this is achieved using the *executed* flag.

- Procedure $aeadv.Byzantine(process)$ simply forwards the call to $sys.Byzantine(process)$.

- Procedure $aeadv.State()$ returns a list of pairs $(\pi \in \Pi_C, m \in \mathscr{M})$ such that π delivered m in *sys*, and π would have delivered m in *sys*, if *adv* was playing instead of *aeadv*.

  This is achieved by querying $sys.State()$, then looping over each element (π, m) of the response. For each (π, m), the procedure loops over every element ρ of $sys.Sample(\pi, m)$, and computes the number n of `Echo`(m, m) messages that π would have received from its echo sample for m in *sys*, if *adv* was playing instead of *aeadv*. This is achieved using the *echoes* table. If n is greater or equal to $\hat{E}$, (π, m) is included in the list returned by the procedure.

- Procedure $aeadv.Sample(process, message)$ returns every process in $sys.Sample(process, message)$ that would have sent an `Echo`($message$, $message'$) message for some message $message'$ to *process* in *sys*, if *adv* was playing instead of *aeadv*. This is achieved using the *echoes* table.

- Procedure $aeadv.Deliver(process, message)$ updates the *echo* table to reflect all the `Echo` messages that *process* will send, as a result of having pb.Delivered *message*. It then forwards the call to $sys.Deliver(process, message)$, causing *process* to pb.Deliver *message*.

- Procedure $aeadv.Echo(process, sample, source, message)$ updates the *echo* table to include the `Echo`($sample$, $message$) message that *process* would receive from *source*, if *adv* was playing instead of *aeadv*.

- Procedure $aeadv.End()$ simply forwards the call to $sys.End()$.

Correctness. We start by proving that no adversary, coupled with Auto-echo decorator, causes the execution to fail.

We start by establishing a preliminary result. Let $\pi \in \Pi_C$, let $m \in \mathcal{M}$. If (π, m) is returned from $aeadv.State()$, then π delivered m in sys. Indeed, (π, m) is returned from $aeadv.State()$ only if (π, m) is returned from $sys.State()$.

Let $\pi \in \Pi_C$, let $m \in \mathcal{M}$. The following hold true:

- An invocation to $aeadv.Step()$ results in one and only one call to $sys.Deliver(\ldots)$, $sys.Echo(\ldots)$ or $sys.End()$. Indeed, if $queue$ is not empty, exactly one call to $sys.Echo(\ldots)$ is issued. Otherwise, $adv.Step()$ is called until $executed =$ `True`, and $executed$ is set to `True` only after an invocation to $sys.Deliver(\ldots)$ or $sys.End()$.
- Procedure $aeadv.State()$ never causes the execution to fail. Indeed, $sys.Sample(\pi, m)$ is called only if (π, m) is returned from $sys.State()$. This means that $sys.Sample(\pi, m)$ is called only if π delivered m in sys. Therefore, $sys.Sample(\pi, m)$ is never invoked from $aeadv.State()$ unless at least one correct process delivered m in sys.
- No invocation of $aeadv.Sample(\ldots)$ causes the execution to fail. Noting that adv is correct, it will never invoke $aeadv.Sample(\pi, m)$ unless (π', m) was returned from a previous invocation of $aeadv.State()$, for some $\pi' \in \Pi_C$. As we previously established, (π', m) is returned from $aeadv.State()$ only if π' delivered m in sys. Therefore, $sys.Sample(\pi, m)$ is never invoked from $aeadv.Sample(\ldots)$ unless at least one correct process delivered m in sys.

Auto-echo. It is easy to prove that Auto-echo decorator always implements an auto-echo adversary. Indeed, every call to $aeadv.Step()$ results in a call to $sys.Echo(\pi, m, \xi, m)$, causing the Byzantine process ξ to send an `Echo`$(m,\ m)$ message to the correct process π, until $queue$ is exhausted.

Therefore, only $sys.Echo(\ldots)$ is invoked until ξ sent an `Echo`$(m,\ m)$ message to π, for every $\pi \in \Pi_C$, every $m \in \mathcal{M}$, and every $\xi \in \Pi \setminus \Pi_C$.

Roadmap. Let $\alpha \in \mathcal{A}$, let $\alpha' = \Delta_{ae}(\alpha)$. Let σ be a system such that α compromises the consistency of σ. Let σ' be an identical copy of σ. In order to prove that α' is more powerful than α, we prove that α' compromises the consistency of σ'.

Trace. We start by noting that, if we couple Auto-echo decorator with σ', we effectively obtain a system instance δ with which α directly exchanges invocations and responses. Here we show that the trace $\tau(\alpha, \sigma)$ is identical to the trace $\tau(\alpha, \delta)$. Intuitively, this means that α has no way of *distinguishing* whether it has been coupled directly with σ, or it has been coupled with σ', with Auto-echo decorator acting as an interface. We prove this by induction.

Let us assume

$$\begin{array}{rcl} \tau(\alpha,\sigma) & = & ((i_1, r_1), \ldots) \\ \tau(\alpha,\delta) & = & ((i'_1, r'_1), \ldots) \\ i_j = i'_j, r_j = r'_j & & \forall j \leq n \end{array}$$

We start by noting that, since α is a deterministic algorithm, we immediately have

$$i_{n+1} = i'_{n+1}$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Byzantine}, \pi)$. Since $aeadv.Byzantine(\pi)$ simply forwards the call to $sys.Byzantine(\pi)$, and σ' is an identical copy of σ, we immediately have $r_{n+1} = r'_{n+1}$.

Before considering the remaining possible values of i_{n+1}, we prove some auxiliary results. Let π be a correct process, let ξ be a Byzantine process, let ρ be a process, let s, m be messages. For every $j \leq n+1$, as we established, we have $i_j = i'_j$. Therefore, after the $(n+1)$-th invocation, the following hold true:

- ρ sent an $\texttt{Echo}(s,\ m)$ message to π in σ if and only if $echoes[\pi][s][\rho] = m$. Indeed, if ρ is correct, ρ sent an $\texttt{Echo}(s,\ m)$ message to π in σ if and only if $aeadv.Deliver(\rho, m)$ was invoked. In turn, $echo[\pi'][s'][\rho]$ was set to m for every $\pi' \in \Pi_C$, $s' \in \mathcal{M}$ if and only if $aeadv.Deliver(\rho, m)$ was invoked. If ρ is Byzantine, ρ sent an $\texttt{Echo}(s,\ m)$ message to π in σ if and only if $aeadv.Echo(\pi, s, \rho, m)$ was invoked. In turn, $echo[\pi][s][\rho]$ was set to m if and only if $aeadv.Echo(\pi, s, \rho, m)$ was invoked.
- If ρ sent an $\texttt{Echo}(m,\ m')$ message to π in σ for some $m' \in \mathcal{M}$, then ρ sent an $\texttt{Echo}(m,\ m'')$ message to π in σ' as well, for some $m'' \in \mathcal{M}$. Indeed, if ρ is correct, then $aeadv.Deliver(\rho, m')$ was invoked. As a result, $sys.Deliver(\rho, m')$ was called, and ρ sent an $\texttt{Echo}(s',\ m')$ message to π' for every $\pi' \in \Pi_C$, $s' \in \mathcal{M}$. If ρ is Byzantine, then it sent an $\texttt{Echo}(m''',\ m''')$ message to π', for every $\pi' \in \Pi_C$, $m''' \in \mathcal{M}$.
- If ρ sent an $\texttt{Echo}(m,\ m)$ message to π in σ, then ρ sent an $\texttt{Echo}(m,\ m)$ message to π in σ' as well. Indeed, if ρ is correct, then $aeadv.Deliver(\rho, m)$ was invoked. As a result, $sys.Deliver(\rho, m)$ was called, and ρ sent an $\texttt{Echo}(s',\ m)$ message to π' for every $\pi' \in \Pi_C$, $s' \in \mathcal{M}$. If ρ is Byzantine, then it sent an $\texttt{Echo}(m',\ m')$ message to π', for every $\pi' \in \Pi_C$, $m' \in \mathcal{M}$.
- If π delivered m in σ, then π delivered m in σ' as well. This follows from the above and the fact that σ' is an identical copy of σ (i.e., π's echo sample for m in σ is identical to π's echo sample in σ'.

Let us assume that $i_{n+1} = (\texttt{State})$. Let π be a correct process, let m be message. We start by noting that $aeadv.State()$ returns (π, m) if and only if π delivered m in σ', and π delivered m in σ. Indeed, (π, m) is added to the return list of $aeadv.State()$ if and only if (π, m) is returned from $sys.State()$, and at least $\hat{E}$ processes sent an $\texttt{Echo}(m,\ m)$ message to π in σ. If $(\pi, m) \in r_{n+1}$, then π delivered m in σ, and π delivered m in σ' as well. Therefore $(\pi, m) \in r'_{n+1}$. If $(\pi, m) \in r'_{n+1}$, then we immediately have that π delivered m in σ, and $(\pi, m) \in r_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Sample}, \pi, m)$. Let ρ be a process. We start by noting that $aeadv.Sample(\pi, m)$ returns ρ if and only ρ sent an $\texttt{Echo}(m,\ m'')$ message to π in σ' for some $m'' \in \mathcal{M}$, and $echoes[\pi][m][\rho] \neq \bot$. If $\rho \in r_{n+1}$, then ρ sent an $\texttt{Echo}(m,\ m')$ message to π in σ, for some $m' \in \mathcal{M}$. Therefore, ρ sent an $\texttt{Echo}(m,\ m'')$ message to π in σ', for some $m'' \in \mathcal{M}$, and $echoes[\pi][m][\rho] = m' \neq \bot$. Consequently, $\rho \in r'_{n+1}$. If $\rho \in r'_{n+1}$, then $echoes[\pi][m][\rho] = m' \neq \bot$ for some $m' \in \mathcal{M}$. Therefore, ρ sent an $\texttt{Echo}(m,\ m')$ message to π in σ, and $\rho \in r_{n+1}$.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \bot = r'_{n+1}$. By induction, we have $\tau(\alpha, \sigma) = \tau(\alpha, \delta)$.

Consistency of σ'. We proved that $\tau(\alpha, \sigma) = \tau(\alpha, \delta)$. Moreover, we proved that if a correct process π eventually delivers a message m in σ, then π also delivers m in σ'.

Since α compromises the consistency of σ, two correct processes π, π' and two distinct messages m, $m' \neq m$ exist such that, in σ, π delivered m and π' delivered m'. Therefore, in σ', π delivered m and π' delivered m'. Therefore α' compromises the consistency of σ'.

Consequently, the adversarial power of α is smaller or equal to the adversarial power of $\alpha' = \Delta_{ae}(a)$, and the lemma is proved. □

3.11.2 Process-sequential adversary

Lemma 21. *The set of process-sequential adversaries $\mathcal{A}_{ps}$ is optimal.*

Proof. We again prove the result using a decorator, i.e., an algorithm that acts as an interface between an adversary and a system. An adversary coupled with a decorator effectively implements an adversary. Here we show that a decorator Δ_{ps} exists such that, for every $\alpha \in \mathcal{A}_{ae}$, the adversary $\alpha' = \Delta_{ps}(\alpha)$ is a process-sequential adversary, and as powerful as α. If this is true, then the lemma is proved: let α^* be an optimal adversary, then the process-sequential $\alpha^+ = \Delta_{ps}(\alpha^*)$ is optimal as well.

Decorator. Algorithm 8 implements **Process-sequential decorator**, a decorator that transforms an auto-echo adversary into a process-sequential adversary. Provided with an auto-echo

Algorithm 8 Process-sequential decorator.

```
1: Implements:
2:     ProcessSequentialAdversary + CobSystem, instance psadv
3:
4: Uses:
5:     AutoEchoAdversary, instance aeadv, system psadv
6:     CobSystem, instance sys
7:
8: procedure psadv.Init() is
9:     perm = {⊥}^C;        cursor = 1;
10:    aeadv.Init();
11:
12: procedure psadv.Step() is
13:    aeadv.Step();
14:
15: procedure psadv.Byzantine(process) is
16:    return sys.Byzantine(process);
17:
18: procedure psadv.State() is
19:    return sys.State();
20:
21: procedure psadv.Sample(process, message) is
22:    sample = ∅;
23:
24:    for all ρ ∈ sys.Sample(process, message) do
25:        if ρ ∈ Π_C then
26:            sample ← sample ∪ {ζ(perm[ζ^{-1}(ρ)])}
27:        else
28:            sample ← sample ∪ {ρ};
29:        end if
30:    end for
31:
32:    return sample;
33:
34: procedure psadv.Deliver(process, message) is
35:    perm[cursor] = ζ^{-1}(process);
36:    sys.Deliver(ζ(cursor), message);
37:    cursor ← cursor + 1;
38:
39: procedure psadv.Echo(process, sample, source, message) is
40:    sys.Echo(process, sample, source, message);
41:
42: procedure psadv.End() is
43:    sys.End();
44:
```

adversary $aeadv$, Process-sequential decorator acts as an interface between $aeadv$ and a system sys, effectively implementing a process-sequential adversary $psadv$. Process-sequential decorator exposes both the adversary and the system interfaces: the underlying adversary $aeadv$ uses $psadv$ as its system.

Process-sequential decorator works as follows:

- Procedure $psadv.Init()$ initializes the following variables:
  - A $perm$ array of C elements: $perm$ is used to consistently translate process identifiers between $aeadv$ and sys.
  - A $cursor$ variable, initially set to 1: at any time, $cursor$ identifies the next process that will pb.Deliver a message in sys.
- Procedure $psadv.Step()$ simply forwards the call to $aeadv.Step()$.
- Procedure $psadv.Byzantine(process)$ simply forwards the call to $sys.Byzantine(process)$.
- Procedure $psadv.State()$ simply forwards the call to $sys.State()$.
- Procedure $psadv.Sample(process, message)$ returns the list of processes returned by $sys.Sample(process, message)$, translated through $perm$. More specifically, for every process ρ in $sys.Sample(process, message)$: if ρ is correct, it is translated to $\zeta(perm[\zeta^{-1}(\rho)])$; if ρ is Byzantine, it is left unchanged.
- Procedure $psadv.Deliver(process, message)$ sets $perm[cursor]$ to $\zeta^{-1}(process)$, then forwards the call to $sys.Deliver(\zeta(cursor), message)$. Finally, it increments $cursor$. This serves the purpose to sequentially cause $\zeta(1)$, $\zeta(2)$, ... to deliver a message, while storing the translation in $perm$ in order for $psadv.Sample(\ldots)$ to provide a response consistent with any previous invocation of $psadv.Deliver(\ldots)$.
- Procedure $psadv.Echo(process, sample, source, message)$ simply forwards the call to $sys.Echo(process, sample, source, message)$.
- Procedure $pasdv.End()$ simply forwards the call to $sys.End()$.

Correctness. We start by proving that no adversary, coupled with Process-sequential decorator, causes the execution to fail.

The following hold true:

- No invocation of $psadv.Sample(\ldots)$ causes the execution to fail. Noting that $aeadv$ is correct, it will never invoke $psadv.Sample(\pi, m)$ unless (π', m) was returned from a

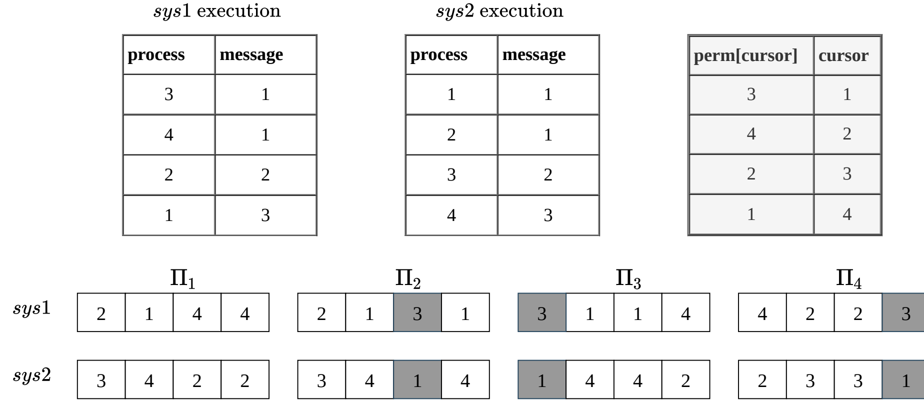


Figure 3.4: Two systems with (one of) their respective echo samples. The table on the right shows the permutation from $sys1$ to $sys2$. Clearly both systems are equally likely. Moreover, the effect of process 3 delivering message 1 (grey) in $sys1$, is equal to process 1 delivering the same message in $sys2$. It can be seen that this holds for all further message deliveries. Intuitively this shows why we can restrict the adversary to always deliver in sequence.

previous invocation of $psadv.State()$, for some $\pi' \in \Pi_C$. Moreover, since $psadv.State()$ simply forwards the call to $sys.State()$, if (pi', m) was returned from $psadv.State()$, then π' delivered m in sys. Therefore, $sys.Sample(\pi, m)$ is never invoked from $psadv.Sample(\ldots)$ unless at least one correct process delivered m in sys.

- Procedure $sys.Sample(\ldots)$ never calls $\zeta(\perp)$. We defer the proof of this result to a later section of this lemma.

- Procedure $sys.Deliver(\ldots)$ is never invoked twice on the same process. Indeed, by definition, ζ is a bijection between $1..C$ and Π_C, and $cursor$ is incremented every time $sys.Deliver(\ldots)$ is called.

Process-sequential. It is easy to prove that Process-sequential decorator always implements a process-sequential adversary. Indeed, $sys.Deliver(\ldots)$ is invoked sequentially on $\zeta(1), \zeta(2), \ldots$ as $cursor$ is incremented, regardless of the process originally provided to $psadv.Deliver(\ldots)$.

System translation. Let α be an adversary. We start by noting that, since α is correct, α always causes every correct process to pb.Deliver a message. We can therefore define a function

$$\mu : \mathscr{A} \times \mathscr{S} \times \Pi_C \to 1..C$$

such that $\mu(\alpha, \sigma, \pi) = d$ if and only if π is the d-th process that α causes to pb.Deliver a message, when α is coupled with σ. We additionally define

$$\left(\mu^{-1}(\alpha, \sigma, d) = \pi\right) \overset{def}{\iff} \left(\mu(\alpha, \sigma, \pi) = d\right)$$

We then define a **system translation function** $\Psi[\alpha] : \mathscr{S} \to \mathscr{S}$ such that, for every system σ, every correct process π, every message m, and every $e \in 1..E$,

$$\Psi[\alpha](\sigma)[\pi][m][e] = \begin{cases} \zeta(\mu(\alpha, \sigma, \sigma[\pi][m][e])) & \text{iff } \sigma[\pi][m][e] \in \Pi_C \\ \sigma[\pi][m][e] & \text{otherwise} \end{cases}$$

Let σ be a system, let $\sigma' = \Psi[\alpha](\sigma)$. Intuitively, σ' is obtained from σ simply by relabeling every correct process in every echo sample. Whenever a correct process π appears in an echo sample in σ, it is replaced with $\zeta(d)$, d being the position of π in the ordered list of processes that α causes to pb.Deliver a message, when coupled with σ. Byzantine processes are left unchanged.

Roadmap. Let $\alpha \in \mathscr{A}_{ae}$, let $\alpha' = \Delta_{ps}(\alpha)$. Let $\sigma \in \mathscr{S}$ such that α compromises the consistency of σ. In order to prove that α' is as powerful as α, we prove that:

- α' compromises the consistency of $\sigma' = \Psi[\alpha](\sigma)$.
- $\Psi[\alpha](\sigma)$ is a permutation on $\mathscr{S}$.

Indeed, if the above are true, then the probability of α' compromising the consistency of a random system σ' is equal to the probability of α compromising the consistency of a random system σ, and the lemma is proved.

Trace. We start by noting that, if we couple Process-sequential decorator with σ', we effectively obtain a system interface δ with which α directly exchanges invocations and responses. Here we show that the trace $\tau(\alpha, \sigma)$ is identical to the trace $\tau(\alpha, \delta)$. Intuitively, this means that α has no way of *distinguishing* whether it has been coupled directly with σ, or it has been coupled with σ', with Process-sequential decorator acting as an interface. We prove this by induction.

Let us assume

$$\begin{aligned} \tau(\alpha, \sigma) &= ((i_1, r_1), \ldots) \\ \tau(\alpha, \delta) &= ((i'_1, r'_1), \ldots) \\ i_j = i'_j, r_j = r'_j & \quad \forall j \le n \end{aligned}$$

with $n \ge 0$ (here $n = 0$ means that this is α's first invocation). We start by noting that, since a is a deterministic algorithm, we immediately have

$$i_{n+1} = i'_{n+1}$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Byzantine}, \pi)$. Let ξ be a Byzantine process. If $\xi \in r_{n+1}$ then, by definition, $\xi \in \sigma[\pi][1]$, i.e., for at least one $e \in 1..E$, $\sigma[\pi][1][e] = \xi$. Therefore, $\sigma'[\pi][1][e] = \xi$, and $\xi \in r'_{n+1}$. If $\xi \notin r_{n+1}$ then, for all $e \in 1..E$, $\sigma[\pi][1][e] \neq \xi$. If $\sigma[\pi][1][e] \in \Pi_C$, then $\sigma'[\pi][1][e] \in \Pi_C$ as well, so $\sigma'[\pi][1][e] \neq \xi$. If $\sigma[\pi][1][e] \in \Pi \setminus \Pi_C$, then $\sigma'[\pi][1][e] = \sigma[\pi][1][e] \neq \xi$. Therefore, $\xi \notin r'_{n+1}$.

Before considering the remaining possible values of i_{n+1}, we prove some auxiliary results. Let π be a correct process, let m be a message, let $d \in 1..C$, let $e \in 1..E$. For every $j \leq n+1$, as we established, we have $i_j = i'_j$. Therefore, after the $(n+1)$-th invocation, the following hold true:

- π pb.Delivered m in σ if and only if $\zeta(\mu(\alpha, \sigma, \pi))$ pb.Delivered m in σ'. Indeed:
  - If π pb.Delivered m in σ, then $psadv.Deliver(\pi, m)$ was invoked. Moreover, by definition, $psadv.Deliver(\pi, m)$ was the $\mu(\alpha, \sigma, \pi)$-th invocation of $psadv.Deliver(\ldots)$. Noting that $cursor$ is incremented at each invocation of $psadv.Deliver(\ldots)$, when $psadv.Deliver(\pi, m)$ was invoked we had $cursor = \mu(\alpha, \sigma, \pi)$. Finally, $psadv.Deliver(\pi, m)$ forwards the call to $sys.Deliver(\zeta(cursor), m)$. Consequently, $\zeta(\mu(\alpha, \sigma, \pi))$ pb.Delivered m in σ'.
  - If $\zeta(\mu(\alpha, \sigma, \pi))$ pb.Delivered m in σ' then $sys.Deliver(\zeta(cursor), m)$ was invoked, with $cursor = \mu(\alpha, \sigma, \pi)$. Noting that $cursor$ is incremented after each invocation of $sys.Deliver(\ldots)$, we have that $psadv.Deliver(\ldots)$ was invoked at least $\mu(\alpha, \sigma, \pi)$ times. By definition, this means that $psadv.Deliver(\pi, m)$ was invoked. Consequently, π pb.Delivered m in σ.
- If π pb.Delivered a message in σ', then $perm[\zeta^{-1}(\pi)] \neq \bot$. Indeed, noting that $cursor$ is incremented every time $psadv.Deliver(\ldots)$ is invoked, we have that ρ pb.Delivered a message in σ' as a result of the $\zeta^{-1}(\pi)$-th invocation of $psadv.Deliver(\ldots)$. As a result, $perm[\zeta^{-1}(\pi)]$ was set to a value other than $\bot$. From this follows that procedure $sys.Sample(\ldots)$ never calls $\zeta(\bot)$.
- If $perm[d] \neq \bot$, then $perm[d] = \zeta^{-1}(\mu^{-1}(\alpha, \sigma, d))$. Indeed, noting that $cursor$ is incremented every time $psadv.Deliver(\ldots)$ is invoked, $perm[d]$ was set to a value other than $\bot$ upon the d-th invocation of $psadv.Deliver(\ldots)$. By the definition of μ, the d-th invocation of $psadv.Deliver(\ldots)$ is $psadv.Deliver(\mu^{-1}(\alpha, \sigma, d), m')$, for some $m' \in \mathcal{M}$.
- $\sigma[\pi][m][e]$ sent an $\texttt{Echo}(m,\ m)$ message to π in σ if and only if $\sigma'[\pi][m][e]$ sent an $\texttt{Echo}(m,\ m)$ message to π in σ'. Indeed, if $\sigma[\pi][m][e] \in \Pi_C$, then $\sigma'[\pi][m][e] = \zeta(\mu(\alpha, \sigma, \sigma[\pi][m][e]))$. Therefore, $\sigma[\pi][m][e]$ pb.Delivered m in σ if and only if $\sigma'[\pi][m][e]$ pb.Delivered m in σ'. Noting that α is an auto-echo adversary, if $\sigma[\pi][m][e] \in \Pi \setminus \Pi_C$, then $\sigma'[\pi][m][e] = \sigma[\pi][m][e]$, and both sent an $\texttt{Echo}(m,\ m)$ message to π (in σ and σ', respectively).

- π delivered m in σ if and only if π delivered m in σ'. This immediately follows from the above.

Let us assume $i_{n+1} = (\texttt{State})$. From the above immediately follows $r_{n+1} = r'_{n+1}$.

Let us assume $i_{n+1} = (\texttt{Sample}, \pi, m)$. Let ρ be a process. The following hold true:

- If $\rho \in r_{n+1}$, then $\rho \in r'_{n+1}$. Indeed, if $\rho \in \Pi_C$, then $\rho \in \sigma[\pi][m]$ and ρ sent an $\texttt{Echo}(m, m')$ message to π in σ, for some $m' \in \mathcal{M}$. Therefore, ρ delivered m' in σ. By definition, $\zeta(\mu(\alpha,\sigma,\rho)) \in \sigma'[\pi][m]$. Moreover, $\zeta(\mu(\alpha,\sigma,\rho))$ delivered m' in σ' and, as a result, it sent an $\texttt{Echo}(m, m')$ message to π in σ'. Therefore, $\zeta(\mu(\alpha,\sigma,\rho)) \in sys.Sample(\pi, m)$. Finally, $perm[\mu(\alpha,\sigma,\rho)] = \zeta^{-1}(\rho)$. Consequently, $\rho \in r'_{n+1}$. If $\rho \in \Pi \setminus \Pi_C$ then $\rho \in \sigma[\pi][m]$ and $\rho \in \sigma'[\pi][m]$. Moreover, ρ sent an $\texttt{Echo}(m, m')$ message to π, for some $m' \in \mathcal{M}$, both in σ and σ'. Consequently, $\rho \in r'_{n+1}$.
- If $\rho \in r'_{n+1}$, then $\rho \in r_{n+1}$. Indeed, if $\rho \in \Pi_C$, then $\zeta(perm^{-1}[\zeta^{-1}(\rho)])$[1] was returned from $sys.Sample(\pi, m)$, in other words $\zeta(perm^{-1}[\zeta^{-1}(\rho)])$ pb.Delivered some message $m' \in \mathcal{M}$ in σ'. Moreover, using our auxiliary result on $perm$ we obtain

  $$\zeta(perm^{-1}[\zeta^{-1}(\rho)]) = \zeta(\mu(\alpha,\sigma,\rho))$$

  therefore $\zeta(\mu(\alpha,\sigma,\rho))$ pb.Delivered m' in σ', and ρ pb.Delivered m' in σ. Finally, since $\zeta(\mu(\alpha,\sigma,\rho)) \in \sigma'[\pi][m]$, then by definition $\rho \in \sigma[\pi][m]$. Consequently, $\rho \in r_{n+1}$.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \bot = r'_{n+1}$. By induction, we have $\tau(\alpha,\sigma) = \tau(\alpha,\delta)$.

Consistency of σ'. We proved that $\tau(\alpha,\sigma) = \tau(\alpha,\delta)$. Moreover, we proved that if a correct process π eventually delivers a message m in σ, then $\zeta(\mu(\alpha,\sigma,\pi))$ also delivers m in σ'.

Since α compromises the consistency of σ, two correct processes π, π' and two distinct messages m, $m' \neq m$ exist such that, in σ, π delivered m and π' delivered m'. Therefore, in σ', $\zeta(\mu(\alpha,\sigma,\pi))$ delivered m and $\zeta(\mu(\alpha,\sigma,\pi'))$ delivered m'. Therefore α' compromises the consistency of σ'.

Translation permutation. We now prove that, for any two σ_a, $\sigma_b \neq \sigma_a$, we have $\Psi[\alpha](\sigma_a) \neq \Psi[\alpha](\sigma_b)$. We prove this by contradiction. Suppose a system σ' exists such that $\sigma' = \Psi[\alpha](\sigma_a) = \Psi[\alpha](\sigma_b)$. We want to prove that $\sigma_a = \sigma_b$.

[1] Noting that $perm$ is injective, we define $perm^{-1}[b] = a \iff perm[a] = b$.

We start by noting that, if $\tau(\alpha,\sigma_a) = \tau(\alpha,\sigma_b)$, then $\sigma_a = \sigma_b$. Indeed, if $\tau(\alpha,\sigma_a) = \tau(\alpha,\sigma_b)$, then for every $\pi \in \Pi_C$ and every $d \in 1..C$ we have

$$\begin{aligned} \mu(\alpha,\sigma_a,\pi) &= \mu(\alpha,\sigma_b,\pi) \\ \mu^{-1}(\alpha,\sigma_a,d) &= \mu^{-1}(\alpha,\sigma_b,d) \end{aligned}$$

and since, by definition, for every $\pi \in \Pi_C$, $m \in \mathcal{M}$ and $e \in 1..E$, we have

$$\begin{aligned} \sigma_a[\pi][m][e] &= \begin{cases} \mu^{-1}(\alpha,\sigma_a,\zeta^{-1}(\sigma'[\pi][m][e])) & \text{iff } \sigma'[\pi][m][e] \in \Pi_C \\ \sigma'[\pi][m][e] & \text{otherwise} \end{cases} \\ \sigma_b[\pi][m][e] &= \begin{cases} \mu^{-1}(\alpha,\sigma_b,\zeta^{-1}(\sigma'[\pi][m][e])) & \text{iff } \sigma'[\pi][m][e] \in \Pi_C \\ \sigma'[\pi][m][e] & \text{otherwise} \end{cases} \end{aligned}$$

we get

$$\sigma_a[\pi][m][e] = \sigma_b[\pi][m][e]$$

and $\sigma_a = \sigma_b$.

We prove that $\tau(\alpha,\sigma_a) = \tau(\alpha,\sigma_b)$ by induction. Let us assume

$$\begin{aligned} \tau(\alpha,\sigma_a) &= ((i_1,r_1),\ldots) \\ \tau(\alpha,\sigma_b) &= ((i'_1,r'_1),\ldots) \\ i_j = i'_j, r_j = r'_j & \quad \forall j \le n \end{aligned}$$

with $n \ge 0$ (here $n = 0$ means that this is α's first invocation). We start by noting that, since a is a deterministic algorithm, we immediately have

$$i_{n+1} = i'_{n+1}$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Byzantine},\pi)$. Let ξ be a Byzantine process. if $\xi \in r_{n+1}$, then for at least one $e \in 1..E$ we have $\sigma_a[\pi][m][e] = \xi$. Therefore, $\sigma'[\pi][m][e] = \xi$, and $\sigma_b[\pi][m][e] = \xi$. Consequently, $\xi \in r'_{n+1}$. The argument can be reversed to prove $\xi \in r'_{n+1} \implies \xi \in r_{n+1}$.

Before considering the remaining possible values of i_{n+1}, we prove some auxiliary result. Let π be a correct process, let m be a message, let $e \in 1..E$. For every $j \le n+1$, as we established, we have $i_j = i'_j$. Therefore, after the $(n+1)$-th invocation, the following hold true:

- π pb.Delivered m in σ_a if and only if π pb.Delivered m in σ_b. Indeed, if π pb.Delivered m in σ_a, then some $j \le (n+1)$ exists such that $i_j = (\texttt{Deliver},\pi,m)$. Since $i'_j = i_j$,

π pb.Delivered m in σ_b as well. The argument can be reversed to prove that, if π pb.Delivered m in σ_b, then π pb.Delivered m in σ_a as well.

- If π pb.Delivered m in σ_a (or, equivalently, σ_b), then $\mu(\alpha, \sigma_a, \pi) = \mu(\alpha, \sigma_b, \pi)$. Indeed, some $j \leq (n+1)$ exists such that $i_j = i'_j = (\texttt{Deliver}, \pi, m)$. Since, for all $h < j$, we also have $i_h = i'_h$, then

$$\begin{aligned} & \left|\left\{h \in 1..(j-1) \mid i_h = (\texttt{Deliver}, \pi' \in \Pi_C, m' \in \mathscr{M})\right\}\right| \\ = & \left|\left\{h \in 1..(j-1) \mid i'_h = (\texttt{Deliver}, \pi' \in \Pi_C, m' \in \mathscr{M})\right\}\right| \end{aligned}$$

- $\sigma_a[\pi][m][e]$ sent an $\texttt{Echo}(m, m)$ message to π in σ_a if and only if $\sigma_b[\pi][m][e]$ sent an $\texttt{Echo}(m, m)$ message to π in σ_b. We prove this by cases:

  - Let us assume that $\sigma_a[\pi][m][e]$ is correct, and pb.Delivered m in σ_a. By definition, we have

$$\begin{aligned} \sigma'[\pi][m][e] &= \zeta(\mu(\alpha, \sigma_a, \sigma_a[\pi][m][e])) \\ \sigma'[\pi][m][e] &= \zeta(\mu(\alpha, \sigma_b, \sigma_b[\pi][m][e])) \end{aligned}$$

    and from the above we have

$$\zeta(\mu(\alpha, \sigma_a, \sigma_a[\pi][m][e])) = \zeta(\mu(\alpha, \sigma_b, \sigma_a[\pi][m][e]))$$

    Equating the two above we get

$$\zeta(\mu(\alpha, \sigma_b, \sigma_a[\pi][m][e])) = \zeta(\mu(\alpha, \sigma_b, \sigma_b[\pi][m][e]))$$

    and noting that μ is always injective, we have $\sigma_a[\pi][m][e] = \sigma_b[\pi][m][e]$. Therefore $\sigma_b[\pi][m][e]$ pb.Delivered m in σ_b.

    The argument can be inverted to prove that, if $\sigma_b[\pi][m][e]$ is correct, and pb.Delivered m in σ_b, then $\sigma_a[\pi][m][e]$ pb.Delivered m in σ_a as well.

  - Let us assume that $\sigma_a[\pi][m][e]$ is correct, but did not pb.Deliver m. From the definition of $\Psi[\alpha]$, we know that $\sigma_b[\pi][m][e]$ is correct as well. By contradiction, following from the above, we have that if $\sigma_b[\pi][m][e]$ pb.Delivered m in σ_b, $\sigma_a[\pi][m][e]$ would have pb.Delivered m in σ_a as well.

    The argument can be inverted to prove that, if $\sigma_b[\pi][m][e]$ is correct, but did not pb.Deliver m in σ_b, then $\sigma_a[\pi][m][e]$ did not pb.Deliver m in σ_a either.

  - Let us assume that $\sigma_a[\pi][m][e]$ is Byzantine. Then, from the definition of $\Psi[\alpha]$, we immediately have $\sigma_b[\pi][m][e] = \sigma_a[\pi][m][e]$ and, since α is an auto-echo adversary, both sent an $\texttt{Echo}(m, m)$ message to π (in their respective systems).

- π delivered m in σ_a if and only if π delivered m in σ_b as well. This follows immediately from the above.

Let us assume $i_{n+1} = (\texttt{State})$. From the above immediately follows $r_{n+1} = r'_{n+1}$.

Let us assume $i_{n+1} = (\texttt{Sample}, \pi, m)$. Let ρ be a process. If $\rho \in r_{n+1}$, then for some $e \in 1..E$, $\sigma_a[\pi][m][e] = \rho$, and ρ sent an $\texttt{Echo}(m, m')$ message to π in σ_a, for some $m' \in \mathscr{M}$. Following from the above, we have $\sigma_b[\pi][m][e] = \rho$ as well, and ρ sent an $\texttt{Echo}(m, m')$ message to π in σ_b as well. Therefore, $\rho \in r'_{n+1}$. The argument can be inverted to prove that, if $\rho \in r'_{n+1}$, then $\rho \in r_{n+1}$ as well.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \perp = r'_{n+1}$. By induction, we have $\tau(\alpha, \sigma_a) = \tau(\alpha, \sigma_b)$.

Therefore, $\sigma_a = \sigma_b$, which contradicts the hypothesis. □

3.11.3 Sequential adversary

Lemma 22. *The set of sequential adversaries $\mathscr{A}_{sq}$ is optimal.*

Proof. We again prove the result using a decorator. Here we show that a decorator Δ_{sq} exists such that, for every $\alpha \in \mathscr{A}_{ps}$, the adversary $\alpha' = \Delta_{sq}(\alpha)$ is a sequential adversary, and as powerful as α. If this is true, then the lemma is proved: let α^* be an optimal adversary, then the sequential $\alpha^+ = \Delta_{sq}(\alpha^*)$ is optimal as well.

Decorator. Algorithm 9 implements **Sequential decorator**, a decorator that transforms a process-sequential adversary into a sequential adversary. Provided with a process-sequential adversary $psadv$, Sequential decorator acts as an interface between $psadv$ and a system sys, effectively implementing a sequential adversary $sqadv$. Sequential decorator exposes both the adversary and the system interfaces: the underlying adversary $psadv$ uses $sqadv$ as its system.

Sequential decorator works as follows:

- Procedure $sqadv.Init()$ initializes the following variables:
  - A $perm$ array of C elements: $perm$ is used to consistently translate messages between $psadv$ and sys.
  - A $cursor$ variable, initially set to 1: at any time, $cursor$ identifies the next message that will be pb.Delivered in sys, if $psadv$ will invoke the delivery of a process whose delivery $psadv$ never invoked before.
  - A $poisoned$ variable: $poisoned$ is set to `True` if and only if at least one correct process in sys is poisoned. This condition is verified by looping over $sys.Byzantine(\pi)$ for every correct process π.

Algorithm 9 Sequential decorator.

```
1: Implements:
2:     SequentialAdversary + CobSystem, instance sqadv
3:
4: Uses:
5:     ProcessSequentialAdversary, instance psadv, system sqadv
6:     CobSystem, instance sys
7:
8: procedure sqadv.Init() is
9:     perm = {⊥}^C;      cursor = 1;      step = 0;
10:
11:    poisoned = False;
12:    for all π ∈ Π_C do
13:        if |sys.Byzantine(π)| ≥ Ê then
14:            poisoned ← True;
15:        end if
16:    end for
17:
18:    psadv.Init();
19:
20: procedure sqadv.Step() is
21:    step ← step + 1;
22:
23:    if poisoned = False or step ≤ (N − C)C^2 then
24:        psadv.Step();
25:    else if step ≤ (N − C)C^2 + C then
26:        sys.Deliver(ζ(step − (N − C)C^2), 1);
27:    else
28:        sys.End();
29:    end if
30:
31: procedure sqadv.Byzantine(process) is
32:    return sys.Byzantine(process);
33:
34: procedure sqadv.State() is
35:    state = ∅;
36:
37:    for all (π, m) ∈ sys.State() do
38:        state ← state ∪ {(π, perm[m])};
39:    end for
40:
41:    return state;
42:
```

```
43: procedure sqadv.Sample(process, message) is
44:     return sys.Sample(process, perm^{-1}[message]);
45:
46: procedure sqadv.Deliver(process, message) is
47:     if message ∈ perm then
48:         sys.Deliver(process, perm^{-1}[message]);
49:     else
50:         perm[cursor] = message;
51:         sys.Deliver(process, cursor);
52:         cursor ← cursor + 1;
53:     end if
54:
55: procedure sqadv.Echo(process, sample, source, message) is
56:     sys.Echo(process, sample, source, message);
57:
58: procedure sqadv.End() is
59:     sys.End();
60:
```

  - A $step$ variable, initially set to 0: at any time, $step$ counts how many times $sqadv.Step()$ has been invoked.

- Procedure $sqadv.Step()$ increments $step$, then implements two different behaviors depending on the value of $poisoned$:

  - If $poisoned = \texttt{True}$, it forwards the call to $psadv.Step()$ for the first $(N-C)C^2$ times. For the next C steps, it sequentially invokes $sys.Deliver(\zeta(1), 1)$, ..., $sys.Deliver(\zeta(C), 1)$. Finally, it calls $sys.End()$.
  - If $poisoned = \texttt{False}$, it forwards the call to $psadv.Step()$.

- Procedure $sqadv.Byzantine(process)$ simply forwards the call to $sys.Byzantine(process)$.

- Procedure $sqadv.State()$ returns the list of process / message pairs returned by $sys.State()$, with each message translated through $perm$. More specifically, $sqadv.State()$ returns $(\pi, perm[m])$ for every (π, m) in $sys.State()$.

- Procedure $sqadv.Sample(process, message)$ simply forwards the call to $sys.Sample(process, perm^{-1}[message])$.

- Procedure $sqadv.Deliver(process, message)$ checks if $psadv$ has already invoked the delivery of $message$ (this is achieved by checking if $message$ is in $perm$). If so, it forwards the call to $sys.Deliver(process, perm^{-1}[message])$. Otherwise, it sets $perm[cursor]$ to $message$, then forwards the call to $sys.Deliver(process, cursor)$. Finally, it increments $cursor$. This mechanism serves two purposes:

- To consistently translate a $sqadv.Deliver(\ldots)$ invocation to a $sys.Deliver(\ldots)$ invocation. More specifically, the set of invocations $psadv.Deliver(\pi_1, m)$, $\ldots$, $psadv.Deliver(\pi_k, m)$ is always translated to $sys.Deliver(\pi_1, m')$, $\ldots$, $sys.Deliver(pi_k, m')$.
- To never cause the pb.Delivery of a message b in sys before every message $a < b$ has been pb.Delivered in sys at least once.

- Procedure $sqadv.Echo(process, sample, source, message)$ simply forwards the call to $sys.Echo(process, sample, source, message)$.
- Procedure $sqadv.End()$ simply forwards the call to $sys.End()$.

Correctness. We start by proving that no adversary, coupled with Sequential decorator, causes the execution to fail. We distinguish two cases, based on the value of $poisoned$.

Let us assume $poisoned = \texttt{True}$. When $sqadv.Step()$ is invoked, the call is forwarded to $psadv.Step()$ only for the first $(N - C)C^2$ times. Noting that $psadv$ is an auto-echo adversary, every call to $psadv.Step()$ results in a call to $sqadv.Echo(\ldots)$. For the next C steps, $sqadv.Step()$ sequentially causes $\zeta(1), \zeta(2), \ldots$ to pb.Deliver message 1. Finally, $sqadv.Step()$ invokes $sys.End()$. Therefore, $sqadv$ never causes the execution to fail, and it implements a process-sequential adversary.

Let us assume $poisoned = \texttt{False}$. Let π be a correct process, let m be a message. The following hold true:

- Procedure $sqadv.State()$ never returns a $(\pi, \bot)$ pair. Indeed, if $(\pi, m) \in sys.State()$, then π pb.Delivered m in sys. Since π is not poisoned, π received at least one $\texttt{Echo}(m, m)$ message from a correct process. Consequently, if (π, m) is returned from $sys.State()$, then at least one correct process pb.Delivered m in sys, i.e., $sys.Deliver(\pi', m)$ was invoked for some $\pi' \in \Pi_C$. The statement is proved by noting that, whenever $sys.Deliver(\pi', m)$ is invoked for some $\pi' \in \Pi_C$, we have $perm[m] \neq \bot$: indeed, either $sys.Sample(process, perm^{-1}[message])$ is invoked, and $message \in perm$, or $sys.Sample(process, cursor)$ is invoked, and $perm[cursor] = message \neq \bot$.
- No invocation of $sqadv.Sample(\ldots)$ causes the execution to fail. Noting that $psadv$ is correct, it will never invoke $sqadv.Sample(\pi, m)$ unless (π', m) was returned from a previous invocation of $sqadv.State()$, for some $\pi' \in \Pi_C$. Since π' is not poisoned, $(\pi', perm^{-1}[m])$ was returned from $sys.State()$, therefore π' delivered $perm^{-1}[m]$ in sys. Therefore $sys.Sample(\pi, m)$ is never invoked from $sqadv.Sample(\ldots)$ unless at least one correct process delivered m in sys.

Sequential. It is easy to prove that Sequential decorator always implements a sequential adversary. Indeed, if $poisoned = \texttt{True}$, $sqadv$ simply causes every correct process to pb.Deliver

message 1 (which trivially implements a sequential adversary). If $poisoned = \texttt{False}$, then whenever $sys.Deliver(\pi, m)$ is invoked, either of the following holds true:

- $m = perm^{-1}[message]$ for some $message \in perm$. In this case $sys.Deliver(\ldots)$ was previously invoked on m (i.e., some process π' exists such that $sys.Deliver(\pi', m)$ was previously invoked).
- $m = cursor$. Then $sys.Deliver(\ldots)$ was never invoked on m. Noting that, whenever $sys.Deliver(\ldots)$ is invoked on a new message, $cursor$ is incremented, we have that every message $l < m$ was previously pb.Delivered by at least one correct process in sys.

System translation. Let α be an adversary. We can define a function

$$\mu : \mathscr{A} \times \mathscr{S} \times \mathscr{M} \rightarrow 1..C \cup \{\bot\}$$

such that:

- $\mu(\alpha, \sigma, m) = (d \in 1..C)$ if and only if m is the d-th distinct message that α causes at least one correct process to pb.Deliver, when α is coupled with σ.
- $\mu(\alpha, \sigma, m) = \bot$ if and only if α never causes any correct process to pb.Deliver m, when α is coupled with σ.

We additionally define $\nu : \mathscr{A} \times \mathscr{S} \rightarrow 1..C$ by

$$\nu(\alpha, \sigma) = \max_{m \in \mathscr{M}} \mu(\alpha, \sigma, m)$$

and

$$\left(\mu^{-1}(\alpha, \sigma, d) = m\right) \overset{def}{\Longleftrightarrow} \left(\mu(\alpha, \sigma, m) = d\right)$$

for all $d \leq \nu(\alpha, \sigma)$. Here $\nu(\alpha, \sigma)$ counts the number of distinct messages that α causes at least one correct process to pb.Deliver, when coupled with σ. It is immediate to see that $\mu(\alpha, \sigma, d) = \bot$ for all $d > \nu(\alpha, \sigma)$.

We then define a **message permutation function** $\chi : \mathscr{A} \times \mathscr{S} \times \mathscr{M} \rightarrow \mathscr{M}$ as follows:

$$\chi(\alpha, \sigma, d) = \begin{cases} \mu^{-1}(\alpha, \sigma, d) & \text{iff } d \leq \nu(\alpha, \sigma) \\ \max m \in \mathscr{M} \mid & \\ \quad \left|\left\{l \leq m : \mu(\alpha, \sigma, l) = \bot\right\}\right| = d - \nu(\alpha, \sigma) & \text{otherwise} \end{cases}$$

For a given α and σ, the permutation χ maps d to the d-th distinct message that is pb.Delivered when α is coupled with σ, if such a message exists. If such a message does not exist, χ simply enumerates sequentially the messages that are never pb.Delivered when α is

coupled with σ.

For example, let us consider the case where $C = 10$ and α coupled with σ causes the pb.Delivery of messages $3, 7, 1, 4$ (in this order of first appearance). Then χ will assume the following values for $d \in 1..C$: $3, 7, 1, 4, 2, 5, 6, 8, 9, 10$.

Finally, we define a **system translation function** $\Psi[\alpha] : \mathscr{S} \to \mathscr{S}$ such that, for system σ, every correct process π and every message m,

$$\Psi[\alpha](\sigma)[\pi][m] = \begin{cases} \sigma[\pi][m] & \text{iff } \exists \pi' \in \Pi_C \mid \pi' \text{ is poisoned in } \sigma \\ \sigma[\pi][\chi(\alpha, \sigma, m)] & \text{otherwise} \end{cases}$$

Let σ be a system, let $\sigma' = \Psi[\alpha](\sigma)$. Intuitively, if at least one correct process is poisoned in σ, then $\sigma' = \sigma$. Otherwise, σ' is obtained from σ by permuting the echo samples of each correct process in σ using χ.

Roadmap. Let $\alpha \in \mathscr{A}_{ps}$, let $\alpha' = \Delta_{sq}(\alpha)$. Let $\sigma \in \mathscr{S}$ such that α compromises the consistency of σ. In order to prove that α' is as powerful as α, we prove that:

- α' compromises the consistency of $\sigma' = \Psi[\alpha](\sigma)$.
- $\Psi[\alpha](\sigma)$ is a permutation on $\mathscr{S}$.

Indeed, if the above are true, then the probability of α' compromising the consistency of a random system σ' is equal to the probability of α compromising the consistency of a random system σ, and the lemma is proved.

Poisoned case. We start by considering the case where $poisoned = $ `True`. Let π be a correct process that is poisoned in σ. Noting that $psadv$ is an auto-echo adversary, π eventually delivers every message. Indeed, every Byzantine process eventually sends to π an `Echo(`m, m`)` message, for every $m \in \mathscr{M}$. Since all of π's echo samples share the same set of at least $\hat{E}$ Byzantine processes, π eventually delivers every message.

As a result, if at least one correct process in σ is poisoned, the consistency of σ is compromised by any auto-echo adversary. Noting that $\sigma' = \Psi[\alpha](\sigma) = \sigma$, and $\Delta_{sq}(\alpha)$ is an auto-echo adversary, we immediately have that α' compromises the consistency of σ' as well.

In the next sections of this proof, we consider the case $poisoned = $ `False`.

Trace. We start by noting that, if we couple Process-sequential decorator with σ', we effectively obtain a system interface δ with which α directly exchanges invocations and responses. Here we show that, if $poisoned = \texttt{False}$, the trace $\tau(\alpha,\sigma)$ is identical to the trace $\tau(\alpha,\delta)$. Intuitively, this means that, if $poisoned = \texttt{False}$, α has no way of *distinguishing* whether it has been coupled directly with σ, or it has been coupled with σ', with Process-sequential decorator acting as an interface. We prove this by induction.

Let us assume $poisoned = \texttt{False}$, and

$$\begin{aligned} \tau(\alpha,\sigma) &= ((i_1, r_1), \ldots) \\ \tau(\alpha,\delta) &= ((i'_1, r'_1), \ldots) \\ i_j = i'_j, r_j = r'_j & \quad \forall j \leq n \end{aligned}$$

with $n \geq 0$ (here $n = 0$ means that this is α's first invocation). We start by noting that, since a is a deterministic algorithm, we immediately have

$$i_{n+1} = i'_{n+1}$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Byzantine}, \pi)$. We can note that $sqadv.Byzantine(process)$ simply forwards the call to $sys.Byzantine(process)$, and χ defines a permutation over $1..C$. Therefore, a message m exists such that π's first echo sample for in σ' is identical to π's echo sample for m in σ. Moreover, all of π's echo samples in σ share the same set of Byzantine processes. Consequently, $r_{n+1} = r'_{n+1}$.

Before considering the remaining possible values of i_{n+1}, we prove some auxiliary results. We start by noting the following:

- Let $d \in 1..C$. At any time, if $perm[d] \neq \bot$, then $perm[d] = \mu^{-1}(\alpha, \sigma, d)$. Indeed, at any time, a message m is in $perm$ if and only if $sqadv.Deliver(\ldots)$ was previously invoked on m. Moreover, whenever $sqadv.Deliver(\ldots)$ is invoked on a message m that is not in $perm$, m is added to $perm$ and $cursor$ is incremented. Therefore $perm[cursor]$ is set to m if and only if $sqadv.Deliver(\ldots)$ was never invoked on m, and $sqadv.Deliver(\ldots)$ was previously invoked on exactly $cursor - 1$ distinct messages. Moreover, by definition, when $sqadv.Deliver(\ldots)$ is invoked on m for the first time, $sqadv.Deliver(\ldots)$ was previously invoked on exactly $\mu(\alpha, \sigma, m) - 1$ distinct messages. Consequently, $cursor = \mu(\alpha, \sigma, m)$, and $m = \mu^{-1}(\alpha, \sigma, cursor)$.
- No two values of $perm$ are equal to each other. Indeed, a message m is added to $perm$ only if $m \notin perm$.
- Let $\pi \in \Pi_C$, let $m \in \mathcal{M}$. If π delivered m, then at least one correct process pb.Delivered m.

This separately holds true both in σ and σ'. Indeed, if π delivered m, then it received at least $\hat{E}$ Echo(m, m) messages from its echo sample for m and, since no correct process is poisoned in neither σ nor σ', at least one of them must have come from a correct process.

Let π be a correct process, let ρ be a process, let m, s be messages. For every $j \leq n+1$, as we established, we have $i_j = i'_j$. By hypothesis, α is an auto-echo adversary, so $i_j = i'_j$, $r_j = r'_j = \perp$ for every $j \leq (N-C)C^2$. Let us consider the non-trivial case $n \geq (N-C)C^2$. After the $(n+1)$-th invocation, the following hold true:

- π pb.Delivered m in σ if and only if π pb.Delivered $\mu(\alpha,\sigma,m)$ in σ'. Indeed:
  - If π pb.Delivered m in σ, then $sqadv.Deliver(\pi,m)$ was invoked.

    If $sqadv.Deliver(\pi,m)$ was the first invocation of $sqadv.Deliver(\ldots)$ on m, then m was not in $perm$, $perm[cursor]$ was set to m, and $sys.Deliver(\pi,cursor)$ was invoked. As we previously proved, however, we have $perm[cursor] = \mu^{-1}(\alpha,\sigma,m)$, so $cursor = \mu(\alpha,\sigma,m)$. Consequently, $sys.Deliver(\pi,\mu(\alpha,\sigma,m))$ was invoked, and π pb.Delivered $\mu(\alpha,\sigma,m)$ in σ'. If $sqadv.Deliver(\pi,m)$ was not the first invocation of $sqadv.Deliver(\ldots)$ on m, then m was in $perm$, and $sys.Deliver(\pi,perm^{-1}(m))$ was invoked. Due to the above, we have again $perm^{-1}[m] = \mu(\alpha,\sigma,m)$. Consequently, $sys.Deliver(\pi,\mu(\alpha,\sigma,m))$ was invoked, and π pb.Delivered $\mu(\alpha,\sigma,m)$ in σ'.
  - If π pb.Delivered $\mu(\alpha,\sigma,m)$ in σ', then $sys.Deliver(\pi,\mu(\alpha,\sigma,m))$ was invoked. If $sys.Deliver(\pi,cursor)$ was invoked, we have that $cursor = \mu(\alpha,\sigma,m)$, and $sqadv.Deliver(\pi,m')$ was invoked for some $m' \notin perm$. As a result, $perm[cursor]$ was set to m'. As we previously established, however,

    $$m' = \mu^{-1}(\alpha,\sigma,cursor) = \mu^{-1}(\alpha,\sigma,\mu(\alpha,\sigma,m)) = m$$

    and $sqadv.Deliver(\pi,m)$ was invoked. As a result, π pb.Delivered m in σ. If $sys.Deliver(\pi,perm^{-1}(m'))$ was invoked for some $m' \in perm$, we have $perm^{-1}[m'] = \mu(\alpha,\sigma,m)$, and again $m' = m$. Consequently, $sqadv.Deliver(\pi,m)$ was invoked, and π pb.Delivered m in σ.
- π received an Echo(m, m) message from ρ in σ if and only if π received an Echo($\mu(\alpha,\sigma,m)$, $\mu(\alpha,\sigma,m)$) message from ρ in σ'. Indeed:
  - If ρ is a correct process, from the above we have that π pb.Delivered m in σ if and only if π pb.Delivered $\mu(\alpha,\sigma,m)$ in σ'. Therefore, ρ sent to π an Echo(m, m) message if and only if ρ sent to π an Echo($\mu(\alpha,\sigma,m)$, $\mu(\alpha,\sigma,m)$) message.
  - If ρ is a Byzantine process then, noting that α is an auto-echo adversary, ρ sent to π an Echo(m, m) both in σ and σ'.

- π received an `Echo`(s, m') message for some $m' \in \mathcal{M}$ from ρ in σ if and only if π received an `Echo`($\mu(\alpha, \sigma, s)$, m'') message for some $m'' \in \mathcal{M}$ from ρ in σ''. Indeed:
  - If ρ is correct, it sent an `Echo`(s', m') message for every $s' \in \mathcal{S}$ and some $m' \in \mathcal{S}$ to π in σ if and only if ρ pb.Delivered a message in σ. Moreover, ρ pb.Delivered a message in σ if and only if ρ pb.Delivered a message in σ'. Finally, ρ pb.Delivered a message in σ' if and only if ρ sent an `Echo`(s'', m'') message for every $s'' \in \mathcal{S}$ and some $m'' \in \mathcal{S}$ to π in σ'.
  - If ρ is Byzantine, it sent an `Echo`(m', m') message for every $m' \in \mathcal{S}$ both in σ and σ'.
- π delivered m in σ if and only if π delivered $\mu(\alpha, \sigma, m)$ in σ'. Indeed, if π delivered m in σ, then at least one correct process pb.Delivered m in σ, and at least one correct process pb.Delivered $\mu(\alpha, \sigma, m)$ in σ'; if π delivered $\mu(\alpha, \sigma, m)$ in σ', then at least one correct process pb.Delivered $\mu(\alpha, \sigma, m)$ in σ', and at least one correct process pb.Delivered m in σ. Following from the definition of χ, π's echo sample for m in σ is identical to π's echo sample for $\mu(\alpha, \sigma, m)$ in σ'. Moreover, π received an `Echo`(m, m) message from ρ in σ if and only if π received an `Echo`($\mu(\alpha, \sigma, m)$, $\mu(\alpha, \sigma, m)$) message from ρ in σ'. Therefore π delivered m in σ if and only if π delivered $\mu(\alpha, \sigma, m)$ in σ'.

Let us assume $i_{n+1} = (\texttt{State})$. Let π be a correct process, let m be a message. The following hold true:

- If $(\pi, m) \in r_{n+1}$, then $(\pi, m) \in r'_{n+1}$. Indeed, π delivered m in σ, therefore π delivered $\mu(\alpha, \sigma, m)$ in σ'. Moreover, $sqadv.Deliver(\ldots)$ was invoked at least once on m, and $perm[\mu(\alpha, \sigma, m)] = m$. Finally, $perm[\mu(\alpha, \sigma, m)]$ was returned from $sqadv.State()$, i.e., $(\pi, m) \in r'_{n+1}$.
- If $(\pi, m) \in r'_{n+1}$, then π delivered $perm^{-1}[m]$ in σ'. Since $perm[m] = \mu^{-1}(\alpha, \sigma, m)$, π delivered $\mu(\alpha, \sigma, m)$ in σ'. Therefore π delivered m in σ, and $(\pi, m) \in r_{n+1}$.

Let us assume $i_{n+1} = (\texttt{Sample}, \pi, m)$. At least one correct process delivered m in σ. Since no correct process is poisoned, at least one correct process pb.Delivered m in σ, and $sqadv.Deliver(\ldots)$ was invoked at least once on m. Therefore, $perm[m] = \mu^{-1}(\alpha, \sigma, m)$. Moreover, from the definition of χ, we have that π's echo sample for m in σ is identical to π's echo sample for $\mu(\alpha, \sigma, m)$ in σ'. Finally, every process that sent an `Echo`(s, m') for some $m' \in \mathcal{M}$ to π in σ sent an `Echo`($\mu(\alpha, \sigma, s)$, m'') for some $m'' \in \mathcal{M}$ to π in σ'. Since $sqadv.Sample(\pi, m)$ forwards the call to $sys.Sample(\pi, perm^{-1}(m))$, we again have $r_{n+1} = r'_{n+1}$.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \bot = r'_{n+1}$. By induction, we have $\tau(\alpha, \sigma) = \tau(\alpha, \delta)$.

Consistency of σ'. We proved that, if $poisoned = \texttt{False}$, then $\tau(\alpha,\sigma) = \tau(\alpha,\delta)$. Moreover, we proved that if a correct process π eventually delivers a message m in σ, then π delivers $\mu(\alpha,\sigma,m)$ in σ'.

Since α compromises the consistency of σ, two correct processes π, π' and two distinct messages m, $m' \neq m$ exist such that, in σ, π delivered m and π' delivered m'. Therefore, in σ', π delivered $\mu(\alpha,\sigma,m)$ and π' delivered $\mu(\alpha,\sigma,m') \neq \mu(\alpha,\sigma,m)$ (since μ is a permutation). Therefore α' compromises the consistency of σ'.

Translation permutation. We now prove that, for any two σ_a, $\sigma_b \neq \sigma_a$, we have $\Psi[\alpha](\sigma_a) \neq \Psi[\alpha](\sigma_b)$. We prove this by contradiction. Suppose a system σ' exists such that $\sigma' = \Psi[\alpha](\sigma_a) = \Psi[\alpha](\sigma_b)$. We want to prove that $\sigma_a = \sigma_b$.

Following from the definition of $\Psi[\alpha]$, if at least one correct process in σ' is poisoned, then we immediately have $\sigma_a = \sigma' = \sigma_b$. Consequently, no correct process in σ' is poisoned.

We start by noting that, if $\tau(\alpha,\sigma_a) = \tau(\alpha,\sigma_b)$, then $\sigma_a = \sigma_b$. Indeed, if $\tau(\alpha,\sigma_a) = \tau(\alpha,\sigma_b)$, then for every $\pi \in \Pi_C$ and every $m \in \mathcal{M}$ we have

$$\mu(\alpha,\sigma_a,m) = \mu(\alpha,\sigma_b,m)$$

from which immediately follows

$$\chi(\alpha,\sigma_a,m) = \chi(\alpha,\sigma_b,m)$$

and, since no correct process is poisoned, for every $\pi \in \Pi_C$ and every $m \in \mathcal{M}$ we have

$$\begin{aligned} \sigma_a[\pi][m] &= \sigma'[\pi][\chi^{-1}(\alpha,\sigma_a,m)] \\ &= \sigma'[\pi][\chi^{-1}(\alpha,\sigma_b,m)] \\ &= \sigma_b[\pi][m] \end{aligned}$$

therefore $\sigma_a = \sigma_b$.

We prove that $\tau(\alpha,\sigma_a) = \tau(\alpha,\sigma_b)$ by induction. Let us assume

$$\begin{aligned} \tau(\alpha,\sigma_a) &= ((i_1,r_1),\ldots) \\ \tau(\alpha,\sigma_b) &= ((i'_1,r'_1),\ldots) \\ i_j = i'_j, r_j = r'_j & \qquad \forall j \le n \end{aligned}$$

with $n \geq 0$ (here $n = 0$ means that this is α's first invocation). We start by noting that, since a is a deterministic algorithm, we immediately have

$$i_{n+1} = i'_{n+1}$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Byzantine}, \pi)$. As we previously established, the Byzantine processes in π's echo samples in σ' are identical to the Byzantine processes in π's echo samples in σ_a and σ_b. Therefore $r_{n+1} = r'_{n+1}$.

Before considering the remaining possible values of i_{n+1}, we prove some auxiliary result. Let π be a correct process, let ρ be a process, let m, s be messages. For every $j \leq n+1$, as we established, we have $i_j = i'_j$. By hypothesis, α is an auto-echo adversary, so $i_j = i'_j$, $r_j = r'_j = \perp$ for every $j \leq (N-C)C^2$. Let us consider the non-trivial case $n \geq (N-C)C^2$. After the $(n+1)$-th invocation, the following hold true:

- ρ sent an $\texttt{Echo}(m, m)$ message to π in σ_a if and only if ρ sent an $\texttt{Echo}(m, m)$ message to π in σ_b. Indeed, if ρ is a correct process, and ρ pb.Delivered m in σ_a, then some $j \leq (n+1)$ exists such that $i_j = (\texttt{Deliver}, \rho, m)$. Since $i'_j = i_j$, ρ pb.Delivered m in σ_b as well. If ρ is Byzantine, and ρ sent an $\texttt{Echo}(m, m)$ message to π in σ_a, then some $j \leq (n+1)$ exists such that $i_j = (\texttt{Echo}, \pi, m, \rho, m)$. Since $i'_j = i_j$, ρ sent an $\texttt{Echo}(m, m)$ message to π in σ_b as well. Both arguments can be reversed to prove that, if ρ sent an $\texttt{Echo}(m, m)$ message to π in σ_b, then ρ sent an $\texttt{Echo}(m, m)$ message to π in σ_a as well.

- ρ sent an $\texttt{Echo}(s, m')$ for some $m' \in \mathcal{M}$ to π in σ_a if and only if ρ sent an $\texttt{Echo}(s, m'')$ message for some $m'' \in \mathcal{M}$ to π in σ_b. Indeed:

  - If ρ is correct, and it sent an $\texttt{Echo}(s, m')$ message to π in σ_a, then it pb.Delivered m' in both σ_a and σ_b. Consequently, ρ sent an $\texttt{Echo}(s, m')$ message to π in σ_b as well. The argument can be inversed to prove that, if ρ is correct and it sent an $\texttt{Echo}(s, m'')$ message for some $m'' \in \mathcal{M}$ to π in σ_b, then ρ sent an $\texttt{Echo}(s, m')$ message for some $m' \in \mathcal{M}$ in σ_a.
  - If ρ is Byzantine, then it sent an $\texttt{Echo}(m', m')$ message for every $m' \in \mathcal{M}$, both in σ and σ'.

- If at least one correct process pb.Delivered m in σ_a (or, equivalently, σ_b), then $\mu(\alpha, \sigma_a, m) = \mu(\alpha, \sigma_b, m)$. Indeed, let j be the minimum index such that $i_j = i'_j = (\texttt{Deliver}, \pi', m)$ for some $\pi' \in \Pi_C$. By definition, we have

$$
\begin{aligned}
\mu(\alpha, \sigma_a, m) &= \left|\{m \in \mathcal{M} \mid \exists k \leq j, \pi' \in \Pi_C : i_k = (\texttt{Deliver}, \pi', m)\}\right| \\
&= \left|\{m \in \mathcal{M} \mid \exists k \leq j, \pi' \in \Pi_C : i'_k = (\texttt{Deliver}, \pi', m)\}\right| \\
&= \mu(\alpha, \sigma_b, m)
\end{aligned}
$$

- π delivered m in σ_a if and only if π delivered m in σ_b. Indeed, if π delivered m in σ_a, then at least one correct process pb.Delivered m both in σ_a and σ_b, and $\mu(\alpha, \sigma_a, m) = \mu(\alpha, \sigma_b, m)$. From the definition of χ, we immediately get $\chi(\alpha, \sigma_a, m) = \chi(\alpha, \sigma_b, m)$ and,

as we previously established, π's echo sample for m in σ_a is identical to π's echo sample for m in σ_b. Since π received the same Echo(m, m) messages in σ_a and σ_b, π delivered m in σ_b as well. The argument can be reversed to prove that, if π delivered m in σ_b, then π delivered m in σ_a as well.

Let us assume $i_{n+1} = (\texttt{State})$. From the above it immediately follows $r_{n+1} = r'_{n+1}$.

Let us assume $i_{n+1} = (\texttt{Sample}, \pi, m)$. As we established, π receives an Echo(m, m') message for some $m' \in \mathcal{M}$ from the same set of processes in σ_a and σ_b. Moreover, since at least one correct process pb.Delivered m in both σ_a and σ_b, π's echo sample for m in σ_a is identical to π's echo sample for m in σ_b. Therefore, $r_{n+1} = r'_{n+1}$.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \bot = r'_{n+1}$. By induction, we have $\tau(\alpha, \sigma_a) = \tau(\alpha, \sigma_b)$.

Therefore, $\sigma_a = \sigma_b$, which contradicts the hypothesis.

□

3.11.4 Non-redundant adversary

Lemma 23. *The set of non-redundant adversaries $\mathcal{A}_{nr}$ is optimal.*

Proof. We again prove the result using a decorator. Here we show that a decorator Δ_{nr} exists such that, for every $\alpha \in \mathcal{A}_{sq}$, the adversary $\alpha' = \Delta_{sq}(\alpha)$ is a non-redundant adversary, and more powerful than α. If this is true, then indeed the lemma is proved: let α^* be an optimal adversary, then the sequential $\alpha^+ = \Delta_{nr}(\alpha^*)$ is optimal as well.

Decorator. Algorithm 10 implements **Non-redundant decorator**, a decorator that transforms a sequential adversary into a non-redundant adversary. Provided with a sequential adversary $sqadv$, Non-redundant decorator acts as an interface between $sqadv$ and a system sys, effectively implementing a non-redundant adversary $nradv$. Non-redundant decorator exposes both the adversary and the system interfaces: the underlying adversary $sqadv$ uses $nradv$ as its system.

Non-redundant decorator works as follows:

- Procedure $nradv.Init()$ initializes a $deliveries$ array that is used to keep track of the message each correct process would have delivered, if $sqadv$ was playing instead of $nradv$.
- Procedure $nradv.Step()$ simply forwards the call to $sqadv.Step()$;

Algorithm 10 Non-redundant decorator.

```
1:  Implements:
2:      NonRedundantAdversary + CobSystem, instance nradv
3:
4:  Uses:
5:      SequentialAdversary, instance sqadv, system nradv
6:      CobSystem, instance sys
7:
8:  procedure nradv.Init() is
9:      deliveries = {⊥}^C;
10:     sqadv.Init();
11:
12: procedure nradv.Step() is
13:     sqadv.Step();
14:
15: procedure nradv.Byzantine(process) is
16:     return sys.Byzantine(process);
17:
18: procedure nradv.State() is
19:     state = ∅;
20:
21:     for all (·, m) ∈ sys.State() do
22:         for all π ∈ Π_C do
23:             n = 0;
24:
25:             for all ρ ∈ sys.Sample(π, m) do
26:                 if ρ ∈ Π \ Π_C or deliveries[ρ] = m then
27:                     n ← n + 1;
28:                 end if
29:             end for
30:
31:             if n ≥ Ê then
32:                 state ← state ∪ {(π, m)};
33:             end if
34:         end for
35:     end for
36:
37:     return state;
38:
39: procedure nradv.Sample(process, message) is
40:     return sys.Sample(process, message);
41:
```

```
42: procedure nradv.Deliver(process, message) is
43:     state = ∅;
44:
45:     for all (·, m) ∈ sys.State() do
46:         state ← state ∪ {m};
47:     end for
48:
49:     if state = {message} then
50:         sys.Deliver(process, message + 1);
51:     else
52:         sys.Deliver(process, message);
53:     end if
54:
55:     deliveries[process] = message;
56:
57: procedure nradv.Echo(process, sample, source, message) is
58:     sys.Echo(process, sample, source, message);
59:
60: procedure nradv.End() is
61:     sys.End();
62:
```

- Procedure $nradv.Byzantine(process)$ simply forwards the call to $sqadv.Byzantine(process)$.

- Procedure $nradv.State()$ returns a list of pairs $(\pi \in \Pi_C, m \in \mathcal{M})$ such at least one correct process delivered m in sys, and π would have delivered m in sys, if $sqadv$ was playing instead of $nradv$.

  This is achieved by querying $sys.State()$, then looping over each message m in the response. For every $\pi \in \Pi_C$, the procedure loops over every element ρ of $sys.Sample(\pi, m)$, and computes the number n of `Echo`(m, m) messages that π would have received from its echo sample for m in sys, if $sqadv$ was playing instead of $nradv$. This is achieved using the $deliveries$ table, and the hypothesis that $sqadv$ is an auto-echo adversary. If n is greater or equal to $\hat{E}$, (π, m) is included in the list returned by the procedure.

- Procedure $nradv.Sample(process, message)$ simply forwards the call to $sys.Sample(process, message)$.

- Procedure $nradv.Deliver(process, message)$ uses $sys.State()$ to determine which messages have been delivered by at least one correct process in sys. If $message$ is the only message that was delivered, the procedure forwards the call to $sys.Deliver(process, message + 1)$. Otherwise, it forwards the call to $sys.Deliver(process, message)$. Finally, it updates the $deliveries$ array to reflect

the fact that $process$ would have pb.Delivered $message$ in sys, if $sqadv$ was playing instead of $nradv$.

- Procedure $nradv.Echo(process, sample, source, message)$ simply forwards the call to $sys.Echo(process, sample, source, message)$.

- Procedure $nradv.End()$ simply forwards the call to $sys.End()$.

Correctness. We start by proving that no adversary, coupled with Non-redundant decorator, causes the execution to fail.

Let $\pi \in \Pi_C$, let $m \in \mathcal{M}$. The following hold true:

- Procedure $nradv.State()$ never causes the execution to fail. Indeed, $sys.Sample(\pi, m)$ is called only if (π', m) was returned from $sys.State()$, for some $\pi' \in \Pi_C$. This means that $sys.Sample(\pi, m)$ is called only if at least one correct process delivered m in sys.

- No invocation of $nradv.Sample(\ldots)$ causes the execution to fail. Noting that $sqadv$ is correct, it will never invoke $nradv.Sample(\pi, m)$ unless (π', m) was returned from a previous invocation of $nradv.State()$, for some $\pi' \in \Pi_C$. Moreover, (π', m) is returned from $nradv.State()$ is and only if, for some $\pi'' \in \Pi_c$, (π'', m) is returned from $sys.State()$. Therefore, $sys.Sample(\pi, m)$ is never invoked unless at least one correct process delivered m in sys.

- Procedure $nradv.Deliver(\ldots)$ never calls $sys.Deliver(\ldots)$ on a message greater than C. Let $m \in \mathcal{M}$. If m is the only message that was delivered in sys, then no correct process is poisoned in sys: indeed, as we proved, if a correct message was poisoned in sys, it would have delivered every message. Therefore, at least one correct process pb.Delivered m. Moreover, since $sqadv$ is a sequential adversary, it invokes $nradv.Deliver(\ldots)$ for the n-th time only on a message $m \leq n$. Since $nradv.Deliver(\ldots)$ is invoked at most C times, we have $n \leq C$, and, since m was delivered as a result of a previous invocation of $nradv.Deliver(\ldots)$, we have $m \leq n - 1$. Consequently, $m + 1 \leq C$.

We further prove that $nradv$ is a sequential adversary. Let $\pi \in \Pi_C$, let $m \in \mathcal{M}$. Since $sqadv$ is sequential, it invokes $nradv.Deliver(\pi, m)$ only if it previously invoked $nradv.Deliver(\ldots)$ on every message $l < m \in \mathcal{M}$. Therefore, $nradv$ can be a non-sequential adversary only as a result of a call to $sys.Deliver(\pi, m+1)$. If m is the only message that was delivered by at least one correct process in sys, then no correct process is poisoned in sys. Therefore, if $sys.Deliver(\pi, m+1)$ is invoked, then, as we established, at least one correct process pb.Delivered m in sys. Noting that the set of messages that are delivered by at least one correct process in sys is non-decreasing, if no correct process is poisoned in sys then $sqadv$ invoked $nradv.Deliver(\ldots)$ on m at least once when no correct process had delivered m. Consequently, for every $l < m$, $nradv.Deliver(\ldots)$, and as a result $sys.Deliver(\ldots)$, was invoked on l.

Non-redundant. It is easy to prove that Non-redundant decorator always implements a non-redundant adversary. Indeed, let $\pi \in \Pi_C$, let $m \in \mathscr{M}$, $sys.Deliver(\pi, m)$ is never invoked if m is the only message that was delivered.

Roadmap. Let $\alpha \in \mathscr{A}_{sq}$, let $\alpha' = \Delta_{nr}(\alpha)$. Let σ be a system such that α compromises the consistency of σ. Let σ' be an identical copy of σ. In order to prove that α' is more powerful than α, we prove that α' compromises the consistency of σ'.

Trace. We start by noting that, if we couple Non-redundant decorator with σ', we effectively obtain a system instance δ with which α directly exchanges invocations and responses. Here we show that the trace $\tau(\alpha, \sigma)$ is identical to the trace $\tau(\alpha, \delta)$. Intuitively, this means that α has no way of *distinguishing* whether it has been coupled directly with σ, or it has been coupled with σ', with Non-redundant decorator acting as an interface. We prove this by induction.

Let us assume

$$
\begin{aligned}
\tau(\alpha, \sigma) &= ((i_1, r_1), \ldots) \\
\tau(\alpha, \delta) &= ((i'_1, r'_1), \ldots) \\
i_j = i'_j, r_j = r'_j & \qquad \forall j \leq n
\end{aligned}
$$

We start by noting that, since α is a deterministic algorithm, we immediately have

$$
i_{n+1} = i'_{n+1}
$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Byzantine}, \pi)$. Since $nradv.Byzantine(\pi)$ simply forwards the call to $sys.Byzantine(\pi)$, and σ' is an identical copy of σ, we immediately have $r_{n+1} = r'_{n+1}$.

Before considering the remaining possible values of i_{n+1}, we prove some auxiliary results. Let π be a correct process, let ρ be a process, let m be a message . For every $j \leq n+1$, as we established, we have $i_j = i'_j$. Therefore, after the $(n+1)$-th invocation, the following hold true:

- π pb.Delivered m in σ if and only if $deliveries[\pi] = m$. This follows immediately from the fact that, whenever $nradv.Deliver(\pi, m)$ is invoked, $deliveries[\pi]$ is set to m.
- π pb.Delivered a message in σ if and only if π pb.Delivered a message in σ'. This follows immediately from the fact that every $nradv.Deliver(\pi, m)$ is always either forwarded to $sys.Deliver(\pi, m)$ or $sys.Deliver(\pi, m+1)$.
- If m was delivered by at least one correct process in σ, then m was delivered by at least

one correct process in σ' as well. Indeed:

- If at least one correct process is poisoned, then it delivered every message both in σ and σ'.
- If no correct process is poisoned then, for some $j^* \leq n+1$, after the j-th invocation, exactly one message m^* was delivered by at least one correct process in σ. This follows from the fact that a non-poisoned process delivers m only as a result of receiving an `Echo`(m, m) message, and no two messages `Echo`(m, m), `Echo`$(m' \neq m, m')$ are ever issued as a result of a single invocation.
- If no correct process is poisoned, and $m = m^*$, then some correct process π^* delivered m in σ as a result of the j^*-th invocation. It is easy to see that, up to the j^*-th invocation, every call to $nradv.Deliver(\pi, m)$ was simply forwarded to $sys.Deliver(\pi, m)$. Therefore, noting that π^*'s echo sample for m is identical in σ and σ', π^* delivered m in sys as well.
- If no correct process is poisoned, and $m \neq m^*$, then no invocation of $nradv.Deliver(\ldots)$ sees m as the only message delivered by at least one correct process in sys. Therefore, all calls to $nradv.Deliver(\pi, m)$ are simply forwarded to $sys.Deliver(\pi, m)$. Consequently, noting that π's echo sample for m is identical in σ and σ', if π delivered m in σ, then π delivered m in σ' as well.

Let us assume that $i_{n+1} = ($`State`$)$. Let π be a correct process, let m be message. The following hold true:

- If $(\pi, m) \in r_{n+1}$, then π delivered m in σ. Therefore, at least one correct process delivered m in σ'. Let π' be a correct process in π's echo sample for m that pb.Delivered m in σ: as we established, we have $deliveries[\pi'] = m$ and π' pb.Delivered a message in σ'. Therefore, $\pi' \in sys.Sample(\pi, m)$. Since $nradv.State(\ldots)$ counts the processes in $sys.Sample(\pi, m)$ that are either Byzantine or have their $deliveries$ value set to m, we have $(\pi, m) \in r'_{n+1}$.
- If $(\pi, m) \notin r_{n+1}$, then less than $\hat{E}$ processes in π's echo sample for m are either Byzantine or have pb.Delivered m. Therefore, less than $\hat{E}$ processes in π's echo sample are either Byzantine or have their $deliveries$ value set to m. Since $sys.Sample(\pi, m)$ is a subset of π's echo sample for m, and since $nradv.State(\ldots)$ counts the processes in $sys.Sample(\pi, m)$ that are either Byzantine or have their $deliveries$ value set to m, $(\pi, m) \notin r'_{n+1}$.

Let us assume that $i_{n+1} = ($`Sample`$, \pi, n)$. By hypothesis, π's echo sample for m is identical in σ and σ'. Moreover, the set of processes that pb.Delivered a message is identical in σ and σ'. Noting that $nradv.Sample(\pi, m)$ simply forwards the call to $sys.Sample(\pi, m)$, we immediately get $r_{n+1} = r'_{n+1}$.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \perp = r'_{n+1}$. By induction, we have $\tau(\alpha, \sigma) = \tau(\alpha, \delta)$.

Consistency of σ'. We proved that $\tau(\alpha, \sigma) = \tau(\alpha, \delta)$. Moreover, we proved that if a message m is eventually delivered by at least a correct process in σ, then m is eventually delivered by at least a correct process in σ' as well.

Since α compromises the consistency of σ, two distinct messages m, $m' \neq m$ exist such that, in σ, both m and m' are delivered by at least one correct process. Therefore, in σ', both m and m' are delivered by at least one correct process as well. Therefore, α' compromises the consistency of σ'.

Consequently, the adversarial power of α is smaller or equal to the adversarial power of $\alpha' = \Delta_{nr}(\alpha)$, and the lemma is proved. □

3.11.5 Sample-blind adversary

Lemma 24. *The set of sample-blind adversaries $\mathscr{A}_{sb}$ is optimal.*

Proof. We again prove the result using a decorator. Here we show that a decorator Δ_{sm} exists such that, for every $\alpha \in \mathscr{A}_{nr}$, the adversary $\alpha' = \Delta_{sm}^{C^2}(\alpha)$ is a sample-blind adversary, and more powerful than α. If this is true, then the lemma is proved: let α^* be an optimal adversary, then the sample-blind $\alpha^+ = \Delta_{sm}^{C^2}(\alpha^*)$ is optimal as well.

Decorator. Algorithm 11 implements **Sample-masking decorator**, a decorator that masks every invocation of $Sample(\pi, m)$ issued by a non-redundant adversary, if $Sample(\pi, m)$ is the first invocation of $Sample(\ldots)$ issued by that adversary.

Provided with a non-redundant adversary $nradv$, Sample-masking decorator acts as an interface between $nradv$ and a system sys. Sample-masking decorator is only guaranteed to mask any invocation to $sys.Sample(\pi, m)$, for one process π and one message m. Noting that $|\Pi_C| = C$ and $|\mathscr{M}| = C$, we have that, for every $\alpha \in \mathscr{A}_{nr}$, $\alpha' = \Delta_{sm}^{C^2}(\alpha)$ is a sample-blind adversary: indeed, all of α's C^2 possible calls to $Sample(\ldots)$ are necessarily masked.

Sample-masking decorator exposes both the adversary and the system interfaces: the underlying adversary $nradv$ uses $smadv$ as its system. Sample-masking decorator works as follows:

- Procedure $smadv.Init()$ initializes the following variables:
  - An $index$ and a $cache$ variable, both initially set to $\perp$: $index$ is used to store the pair $(\pi \in \Pi_C, m \in \mathscr{M})$ that was provided as argument to the first invocation of

Algorithm 11 Sample-masking decorator.

```
 1: Implements:
 2:     SampleMaskedAdversary + CobSystem, instance smadv
 3:
 4: Uses:
 5:     NonRedundantAdversary, instance nradv, system smadv
 6:     CobSystem, instance sys
 7:
 8: procedure smadv.Init() is
 9:     index = ⊥;          cache = ⊥;
10:     trace = [];
11:     deliveries = {⊥}^C;
12:     nradv.Init();
13:
14: procedure optimize(process, message) is
15:     smadv.Byzantine(process);
16:     best.sample = ⊥;     best.probability = 0;
17:
18:     for all sample ∈ Π^E do
19:         systems = 0;
20:         compromissions = 0;
21:         for all σ ∈ 𝒮 do
22:             if trace ~ σ and σ[process][message] = sample then
23:                 systems ← systems + 1;
24:                 if NonRedundantAdversary ↘ σ then
25:                     compromissions ← compromissions + 1;
26:                 end if
27:             end if
28:         end for
29:
30:         if systems > 0 and compromissions/systems > best.probability then
31:             best.sample = sample;
32:             best.probability = compromissions/systems;
33:         end if
34:     end for
35:
36:     return best.sample;
37:
38: procedure smadv.Step() is
39:     nradv.Step();
40:
41: procedure smadv.Byzantine(process) is
42:     trace ← trace + [(Byzantine, process, sys.Byzantine(process))];
43:     return sys.Byzantine(process);
44:
```

```
45: procedure smadv.State() is
46:     trace ← trace + [(State, sys.State())];
47:     state = sys.State() \ {index};
48:
49:     if index ≠ ⊥ then
50:         (π, m) = index;
51:
52:         n = 0;
53:         for all ρ ∈ cache do
54:             if ρ ∈ Π \ Π_C or deliveries[ρ] = m then
55:                 n ← n + 1;
56:             end if
57:         end for
58:
59:         if n ≥ Ê then
60:             state ← state ∪ {(π, m)};
61:         end if
62:     end if
63:
64:     return state;
65:
66: procedure smadv.Sample(process, message) is
67:     if index = ⊥ then
68:         index ← (process, message);
69:         cache ← optimize(process, message);
70:     end if
71:
72:     if (process, message) = index then
73:         sample = {};
74:         for all π ∈ cache do
75:             if deliveries[π] ≠ ⊥ then
76:                 sample ← sample ∪ {π};
77:             end if
78:         end for
79:         return sample;
80:     else
81:         return sys.Sample(process, message);
82:     end if
83:
84: procedure smadv.Deliver(process, message) is
85:     trace ← trace + [(Deliver, (process, message))];
86:     deliveries[process] = message;
87:     sys.Deliver(process, message);
88:
```

```
89: procedure smadv.Echo(process, sample, source, message) is
90:     trace ← trace + [(Echo, (process, sample, source, message))];
91:     sys.Echo(process, sample, source, message);
92:
93: procedure smadv.End() is
94:     sys.End();
95:
```

$smadv.Sample(\ldots)$; $cache$ is used to store the content of the echo sample $smadv$ generates for (π, m) when $smadv.Sample(\ldots)$ is invoked for the first time. This guarantees that subsequent invocations of $smadv.Sample(\pi, m)$ are provided with consistent responses throughout the entire adversarial execution.

– A $trace$ array: $trace$ is used to store the sequence of invocations and responses exchanged between $nradv$ and sys.

– A $deliveries$ array of C elements: $deliveries$ is used to track the message pb.Delivered in sys by each correct process.

- Procedure $optimize(process, message)$ returns the sample $sample$ for $(process, message)$ that maximizes the probability of $nradv$ winning against a random system σ that is compatible with $trace$, and satisfies $\sigma[process][message] = sample$. This is achieved as follows:

  – The procedure calls $smadv.Byzantine(process)$, causing an invocation to $sys.Byzantine(process)$ to be appended to $trace$ along with its response. This is necessary because, if $smadv.Byzantine(process)$ was never invoked before, the set of Byzantine processes in the generated $sample$ might differ from the Byzantine processes in $process$' echo sample for $message$ in sys. Noting that all of $process$' echo samples in sys share the same set of Byzantine processes, a subsequent call to $smadv.Byzantine(process)$ could return a set of Byzantine processes that is inconsistent with the $sample$, causing undefined behavior on $nradv$.

  – The procedure loops over every possible value of $sample$. For each value of $sample$, it counts the number $systems$ of systems σ that are compatible with $trace$, and satisfy $\sigma[process][message] = sample$. Among the systems that satisfy those two constraints, the procedure counts the number $compromissions$ of systems whose consistency the adversary would compromise.

  – The procedure returns the value of $sample$ that satisfies $systems > 0$, and maximizes $compromissions/systems$. In other words, the procedure returns a sample $sample$ that is compatible with at least with one of the systems that are compatible with $trace$, and maximizes the probability that the adversary would compromise the consistency of a randomly selected system compatible with $trace$, picked among those where $process$' echo sample for $message$ is $sample$.

- Procedure $smadv.Byzantine(process)$ appends to $trace$ the invocation of $sys.Byzantine(process)$ along with its response. It then forwards the call to $sys.Byzantine(process)$.
- Procedure $smadv.State()$ appends to $trace$ the invocation of $sys.State()$ along with its response. It then returns the response of $sys.State()$, modified to be compatible with any previous masked invocation of $sys.Sample(\ldots)$. More specifically, if $index = (\pi, m) \neq \bot$ (i.e., $nradv$'s first invocation of $smadv.Sample(\ldots)$ was $smadv.Sample(\pi, m)$), then (π, m) is included in the set of pairs returned by $smadv.State()$ only if π would have delivered m in sys, if π's echo sample for m was $cache$. This is achieved by looping over every process in $cache$, and counting the number n of those processes that are either Byzantine, or pb.Delivered m (this is achieved using the $deliveries$ array).
- Procedure $smadv.Sample(process, message)$ determines whether $smadv.Sample(\ldots)$ has ever been invoked before by checking the value of $index$. If it has not, it sets $index$ to $(process, message)$, and generates a sample for $(process, message)$ by setting $cache$ to the value returned by $optimize(process, sample)$.

  If $(process, message)$ is equal to $index$, the procedure returns the set of processes in $cache$ that pb.Delivered a message in sys. This is achieved by looping over every process ρ in $cache$, and adding ρ to the response if ρ is either Byzantine, or satisfy $deliveries[\rho] \neq \bot$.

  If $(process, message)$ is not equal to $index$, the call is forwarded to $sys.Sample(process, message)$.
- Procedure $smadv.Deliver(process, message)$ appends to $trace$ the invocation of $sys.Deliver(process, message)$. To reflect the fact that $process$ pb.Delivered m in sys, it then updates the $deliveries$ array. Finally, it forwards the call to $sys.Deliver(process, message)$.
- Procedure $smadv.Echo(process, sample, source, message)$ appends to $trace$ the invocation of $sys.Echo(process, sample, source, message)$. It then forwards the call to $sys.Echo(process, sample, source, message)$.
- Procedure $smadv.End()$ simply forwards the call to $sys.End()$.

Correctness. We start by proving that no adversary has undefined behavior when coupled with `Sample-masked decorator`. An adversary has undefined behavior if, at any point, the sequence of invocations and responses it exchanges with $smadv$ is incompatible with every system.

Let $\pi \in \Pi_C$, let $m \in \mathcal{M}$, let us assume that the first invocation to $smadv.Sample(\ldots)$ is $smadv.Sample(\pi, m)$. We start by noting that every invocation in $smadv$ is forwarded to

the corresponding invocation in sys except for $smadv.State()$ and $smadv.Sample(\pi, m)$. Moreover, before the first invocation of $smadv.Sample(\pi, m)$, $index$ is set to $\bot$ and, as a result, $smadv.State()$ effectively forwards to $sys.State()$. Therefore, the trace exchanged between $nradv$ and $smadv$ is trivially compatible with sys before the first invocation of $smadv.Sample(\pi, m)$.

When $smadv.Sample(\pi, m)$ is invoked for the first time, $cache$ is set to $optimize(\pi, m)$. When $optimize(\pi, m)$ is called, it calls $smadv.Byzantine(\pi)$, which appends the invocation and the corresponding response to $trace$. After that, the set of systems that are compatible with $trace$ is non empty, as it trivially includes sys. The procedure $optimize(\pi, m)$ returns a sample $sample$ only if at least one system σ is compatible with $trace$, and satisfies $\sigma[\pi][m] = sample$. Since $sys.Byzantine(\pi)$ is in $trace$, the Byzantine component of $sample$ is identical to $sys.Byzantine(\pi)$: indeed, any system σ where the Byzantine component of $\sigma[\pi][m]$ is different from $sys.Byzantine(\pi)$ is incompatible with σ.

Therefore, the system obtained by replacing π's echo sample for m in sys with $cache$ is a valid system, and it is compatible with $trace$ up to the first invocation of $smadv.Sample(\pi, m)$. Moreover, $trace$ will always be compatible with such system. Indeed:

- Every subsequent call to $smadv.Sample(\pi, m)$ uses the $deliveries$ table to determine which processes in $cache$ pb.Delivered a message in sys, thus returning a response that is consistent with π's echo sample for m being $cache$.

- Every subsequent call to $smadv.State()$ includes (π, m) in its response only if at least $\hat{E}$ processes in $cache$ are either Byzantine or pb.Delivered m in sys (this is verified using the $deliveries$ table).

This proves that that no adversary, coupled with `Sample-masked decorator`, has undefined behavior.

Sample-blind. It is easy to see that Sample-masking decorator masks the first invocation to $Sample(\ldots)$ issued by the decorated adversary. Indeed, if $smadv.Sample(\pi, m)$ is the first invocation of $smadv.Sample(\ldots)$ issued by $nradv$, then $index$ is set to (π, m), and $sys.Sample(\pi, m)$ is never be invoked.

Let α be a non-redundant adversary, we have that $\Delta_{sb}(\alpha)$ issues calls to $Sample(\ldots)$ for at most $C^2 - 1$ pairs $(\pi' \in \Pi_C, m' \in \mathcal{M})$. The same argument can be applied again to see that, by composing Sample-masking decorator with itself C^2 times, all possible calls to $Sample(\ldots)$ are masked. Therefore, $\alpha' = \Delta_{sb}^{C^2}(\alpha)$ is a sample-blind adversary.

Sample replacement. Let α be an adversary, let σ be a system. We define a function $\nu : \mathscr{A} \times \mathscr{S} \to \mathbb{N} \cup \{\bot\}$ by

$$\nu(\alpha,\sigma) = \min n \mid \left(\tau(\alpha,\sigma)_n = ((\texttt{Sample}, \pi \in \Pi_C, m \in \mathscr{M}), \bot)\right)$$

Intuitively, $\nu(\alpha,\sigma)$ returns the index of the first invocation of $Sample(\ldots)$ in $\tau(\alpha,\sigma)$ if such invocation exists, and $\bot$ otherwise. We additionally define $\pi(\alpha,\sigma)$ and $m(\alpha,\sigma)$ by

$$\tau(\alpha,\sigma)_{\nu(\alpha,\sigma)} = ((\texttt{Sample}, \pi(\alpha,\sigma), m(\alpha,\sigma)), \bot)$$

if $\nu(\alpha,\sigma) \neq \bot$, and by

$$\pi(\alpha,\sigma) = m(\alpha,\sigma) = \bot$$

if $\nu(\alpha,\sigma) = \bot$. Whenever at least an invocation to $Sample(\ldots)$ is issued when α is coupled with σ, $\pi(\alpha,\sigma)$ and $m(\alpha,\sigma)$ are the arguments to that invocation.

We then define $\sigma^- : \mathscr{A} \times \mathscr{S} \to \mathbb{N} \cup \{\bot\}$, $\sigma^+ : \mathscr{A} \times \mathscr{S} \to \mathbb{N} \cup \{\bot\}$. If $\nu(\alpha,\sigma) \neq \bot$

$$\begin{aligned} \sigma^-(\alpha,\sigma) &= \max n < \nu(\alpha,\sigma) \mid \tau(\alpha,\sigma)_n = ((\texttt{State}), r_n), \cancel{\Psi(r_n,\alpha,\sigma)} \\ \sigma^+(\alpha,\sigma) &= \min n < \nu(\alpha,\sigma) \mid \tau(\alpha,\sigma)_n = ((\texttt{State}), r_n), \Psi(r_n,\alpha,\sigma) \end{aligned}$$

where Ψ is a predicate defined as

$$\Psi(r_n,\alpha,\sigma) = (\pi(\alpha,\sigma), m(\alpha,\sigma)) \in r_n$$

Otherwise, i.e. if $\nu(\alpha,\sigma) = \bot$

$$\sigma^-(\alpha,\sigma) = \sigma^+(\alpha,\sigma) = \bot$$

otherwise. Intuitively, when $\nu(\alpha,\sigma) \neq \bot$: $\sigma^-(\alpha,\sigma)$ returns the index of the last invocation of $State()$ prior to $\nu(\alpha,\sigma)$ that did not include $(\pi(\alpha,\sigma), m(\alpha,\sigma))$ in its response; $\sigma^+(\alpha,\sigma)$ returns the index of the first invocation of $State()$ prior to $\nu(\alpha,\sigma)$ that included $(\pi(\alpha,\sigma), m(\alpha,\sigma))$ in its response.

We additionally define $\delta : \mathscr{A} \times \mathscr{S} \times \mathscr{M} \times \mathbb{N} \to \mathbb{P}(\Pi_C)$ by

$$\pi \in \delta(\alpha,\sigma,m,n) \overset{def}{\iff} \exists j < n \mid \tau(\alpha,\sigma)_j = ((\texttt{Deliver}, \pi, m), \bot) \tag{3.9}$$

Intuitively, π is in $\delta(\alpha,\sigma,m,n)$ if α invokes $Deliver(\pi, m)$ before the n-th invocation it issues, when coupled with σ. In other words, $\delta(\alpha,\sigma,m,n)$ represents the set of correct processes that pb.Deliver m before the n-th invocation, when α is coupled with σ.

Finally, we define $\delta^- : \mathscr{A} \times \mathscr{S} \rightarrow \mathbb{P}(\Pi_C)$, $\delta^+ : \mathscr{A} \times \mathscr{S} \rightarrow \mathbb{P}(\Pi_C)$ by

$$\delta^-(\alpha,\sigma) = \begin{cases} \delta(\alpha,\sigma,m(\alpha,\sigma),\sigma^-(\alpha,\sigma)) & \text{iff } \sigma^-(\alpha,\sigma) \neq \bot \\ \emptyset & \text{otherwise} \end{cases}$$

$$\delta^+(\alpha,\sigma) = \begin{cases} \delta(\alpha,\sigma,m(\alpha,\sigma),\sigma^+(\alpha,\sigma)) & \text{iff } \sigma^+(\alpha,\sigma) \neq \bot \\ \Pi_C & \text{otherwise} \end{cases}$$

Intuitively:

- When $\sigma^-(\alpha,\sigma) \neq \bot$, $\delta^-(\alpha,\sigma)$ represents the set of processes that pb.Delivered $m(\alpha,\sigma)$ before $\sigma^-(\alpha,\sigma)$. Intuitively, δ^- is designed to guarantee that less than $\hat{E}$ elements of $\sigma[\pi(\alpha,\sigma)][m(\alpha,\sigma)]$ are either Byzantine or included in $\delta^-(\alpha,\sigma)$. If this was not the case, the $\sigma^-(\alpha,\sigma)$-th invocation of $State()$ would have included $(\pi(\alpha,\sigma),m(\alpha,\sigma))$ in its response.

- When $\sigma^+(\alpha,\sigma) \neq \bot$, $\delta^+(\alpha,\sigma)$ represents the set of processes that pb.Delivered $m(\alpha,\sigma)$ before $\sigma^+(\alpha,\sigma)$. Intuitively, δ^+ is designed to guarantee that at least $\hat{E}$ elements of $\sigma[\pi(\alpha,\sigma)][m(\alpha,\sigma)]$ are either Byzantine or included in $\delta^+(\alpha,\sigma)$. If this was not then case, the $\sigma^+(\alpha,\sigma)$-th invocation of $State()$ would not have included $(\pi(\alpha,\sigma),m(\alpha,\sigma))$ in its response.

All the above definitions allow us to define a **sample replacement function** $\mathscr{E}[\alpha] : \mathscr{S} \rightarrow \mathbb{P}\left(\Pi^E\right)$ by

$$\mathscr{E}[\alpha](\sigma) = \emptyset$$

if $\nu(\alpha,\sigma) = \bot$ and

$$\bar{E} \in \mathscr{E}[\alpha](\sigma) \overset{def}{\Longleftrightarrow} \begin{cases} \sigma[\pi(\alpha,\sigma)][m(\alpha,\sigma)][n] \in \Pi \setminus \Pi_C \implies \\ \quad (\bar{E}[n] = \sigma[\pi(\alpha,\sigma)][m(\alpha,\sigma)][n]) \\ \left|\left\{n \in 1..E \mid \bar{E}[n] \in \delta^-(\alpha,\sigma) \cup (\Pi \setminus \Pi_C)\right\}\right| < \hat{E} \\ \left|\left\{n \in 1..E \mid \bar{E}[n] \in \delta^+(\alpha,\sigma) \cup (\Pi \setminus \Pi_C)\right\}\right| \geq \hat{E} \end{cases}$$

otherwise. Intuitively, $\mathscr{E}[\alpha]$ is designed so that, if α is non-redundant, when $\nu(\alpha,\sigma) \neq \bot$, a sample E is in $\mathscr{E}[\alpha](\sigma)$ if, by replacing $\pi(\alpha,\sigma)$'s echo sample for $m(\alpha,\sigma)$ in σ with E, we obtain a system σ' that is *interchangeable* with σ, i.e., a system that cannot be distinguished from σ up to the $\nu(\alpha,\sigma)$-th invocation, and whose consistency is compromised by the same set of traces. We prove these two properties in the next section of this proof.

More specifically, a sample $\bar{E}$ is in $\mathscr{E}[\alpha](\sigma)$ if it satisfies the following conditions:

- $\bar{E}$ shares the set of Byzantine processes in $\sigma[\pi(\alpha,\sigma)][m(\alpha,\sigma)]$.

- Less than $\hat{E}$ processes in $\bar{E}$ pb.Deliver $m(\alpha,\sigma)$ before the last invocation of $State(\ldots)$ in $\tau(\alpha,\sigma)$ (before $\nu(\alpha,\sigma)$) that does not include $(\pi(\alpha,\sigma), m(\alpha,\sigma))$ in its response.

- At least $\hat{E}$ processes in $\bar{E}$ pb.Deliver $m(\alpha,\sigma)$ before the first invocation of $State(\ldots)$ in $\tau(\alpha,\sigma)$ (before $\nu(\alpha,\sigma)$) that includes $(\pi(\alpha,\sigma), m(\alpha,\sigma))$ in its response.

Sample interchangeability. Let α be a non-redundant adversary, let σ be a system such that $\nu(\alpha,\sigma) \neq \perp$. Let $\pi^* = \pi(\alpha,\sigma)$, let $m^* = m(\alpha,\sigma)$. Let σ' be a system such that, for every pair $(\pi, m) \neq (\pi^*, m^*)$ (i.e., $\pi \neq \pi^*$ or $m \neq m^*$), the two following statements hold:

$$\begin{aligned} \sigma'[\pi^*][m^*] &\in \mathscr{E}[\alpha](\sigma) \\ \sigma'[\pi][m] &= \sigma[\pi][m] \end{aligned}$$

In this section, we prove the following:

$$\forall n < \nu(\alpha,\sigma), \tau(\alpha,\sigma)_n = \tau(\alpha,\sigma')_n$$
$$(\alpha \searrow \sigma) \implies (\tau(\alpha,\sigma) \searrow \sigma')$$

We establish the first result by induction. Let us assume

$$\begin{aligned} \tau(\alpha,\sigma) &= ((i_1, r_1), \ldots) \\ \tau(\alpha,\sigma') &= ((i'_1, r'_1), \ldots) \\ i_j = i'_j, r_j = r'_j & \quad \forall j \leq n \end{aligned}$$

with $n \geq 0$ (here $n = 0$ means that this is α's first invocation). We start by noting that, since a is a deterministic algorithm, we immediately have

$$i_{n+1} = i'_{n+1}$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us consider the case $i_{n+1} = (\texttt{Byzantine}, \pi, m)$. Following from the definition of $\mathscr{E}[\alpha](\sigma)$, π^*'s echo sample for m^* in σ' includes the same set of Byzantine processes as π^*'s echo sample for m^* in σ. Since all other echo samples are trivially identical in σ and σ', we have $r_{n+1} = r'_{n+1}$.

Let us consider the case $i_{n+1} = (\texttt{State})$. Let $\pi \in \Pi_C$, let $\rho \in \Pi$, let $m \in \mathscr{M}$. Noting that $i_j = i'_j \; \forall j \leq n+1$, we trivially have that ρ sent an Echo(m, m) message to π in σ if and only if ρ sent an Echo(m, m) message to π in σ. Noting that all echo samples but π^*'s echo sample for m^* are identical in σ, we immediately get that the symmetric difference between r_{n+1} and r'_{n+1} can only include (π^*, m^*). The following hold true:

- If $(\pi^*, m^*) \in r_{n+1}$, then $(\pi^*, m^*) \in r'_{n+1}$. Indeed, if $(\pi^*, m^*) \in r_{n+1}$, then by definition $\sigma^+(\alpha,\sigma) \leq n+1$. Therefore, by definition, every correct process in $\delta^+(\alpha,\sigma)$ pb.Delivered m^* (both in σ and σ'). Noting that α is an auto-echo adversary, every process in $\delta^+(\alpha,\sigma) \cup (\Pi \setminus \Pi_C)$ sent an $\texttt{Echo}(m^*,\ m^*)$ message to π^*, both in σ and σ'. Finally, by definition, $\mathscr{E}[\alpha](\sigma)$ includes at least $\hat{E}$ processes in $\delta^+(\alpha,\sigma) \cup (\Pi \setminus \Pi_C)$. Therefore π^* delivered m^* in σ', and $(\pi^*, m^*) \in r_{n+1}$.

- If $(\pi^*, m^*) \notin r_{n+1}$, then $(\pi^*, m^*) \notin r'_{n+1}$. Indeed, if $(\pi^*, m^*) \in r_{n+1}$, then by definition $\sigma^-(\alpha,\sigma) \geq n+1$. Therefore, by definition, every correct process that pb.Delivered m^* (both in σ and σ') is included in $\delta^-(\alpha,\sigma)$. Finally, by definition, $\mathscr{E}[\alpha](\sigma)$ includes less than $\hat{E}$ processes in $\delta^-(\alpha,\sigma) \cup (\Pi \setminus \Pi_C)$. Therefore π^* did not deliver m^* in σ', and $(\pi^*, m^*) \notin r_{n+1}$.

which proves $r_{n+1} = r'_{n+1}$.

Noting that, by definition, $n < \nu(\alpha,\sigma)$, i_{n+1} cannot be $(\texttt{Sample}, \pi, m)$.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \bot = r'_{n+1}$. By induction, we have

$$\forall n < \nu(\alpha,\sigma), \tau(\alpha,\sigma) = \tau(\alpha,\sigma')$$

Let us assume that α compromises the consistency of σ. We want to prove that $\tau(\alpha,\sigma)$ compromises the consistency of σ'.

We start by noting that, since by definition α's $\nu(\alpha,\sigma)$-th invocation in $\tau(\alpha,\sigma)$ is $(\texttt{Sample}, \pi^*, m^*)$ then, since α is correct, for some $j < \nu(\alpha,\sigma)$, the j-th invocation in $\tau(\alpha,\sigma)$ is $(\texttt{State})$, and its response includes (π, m^*) for some $\pi \in \Pi_C$. Therefore, before the $\nu(\alpha,\sigma)$-th invocation, at least one correct process in σ delivered m^*.

We previously proved, however, that since $j < \nu(\alpha,\sigma)$, we have $\tau(\alpha,\sigma)_j = \tau(\alpha,\sigma')_j$. Therefore, at least one correct process delivered m^* in σ' as well.

Since α compromises the consistency of σ, at least one correct process π' eventually delivers a message $m' \neq m^*$ in σ. Noting that π''s echo sample for m' is identical in σ and σ', we immediately have that π' delivers m' in σ' as well.

System optimization. Let α be a non-redundant adversary, let σ be a system. In the previous section of this proof, we proved that, if we replace $\pi(\alpha,\sigma)$'s echo sample for $m(\alpha,\sigma)$ in σ with any sample in $\mathscr{E}[\alpha](\sigma)$, we obtain a system σ' such that $\tau(\alpha,\sigma)_n = \tau(\alpha,\sigma')_n$ for all $n < \nu(\alpha,\sigma)$.

We start by defining a function $\mathscr{N} : \mathscr{A} \rightarrow \mathbb{P}(\mathscr{S})$ by

$$\mathscr{N}(\alpha) = \{\sigma \in \mathscr{S} \mid \nu(\alpha,\sigma) \neq \bot\}$$

Provided with an adversary α, $\mathscr{N}$ returns the set of systems coupled with which α issues at least one invocation to $Sample(\ldots)$.

We then define a function $\mathscr{S}[\alpha] : \mathscr{N}(\alpha) \to \mathbb{P}(\mathscr{N}(\alpha))$ by

$$\mathscr{S}[\alpha](\sigma) = \left\{\sigma' \in \mathscr{S} \mid \tau(\alpha,\sigma)_{1..(\nu(\alpha,\sigma)-1)} \sim \sigma'\right\}$$

Intuitively, when $\nu(\alpha,\sigma) \neq \perp$, $(S)[\alpha](\sigma)$ returns the set of systems that α cannot distinguish from σ, before the first invocation of $Sample(\ldots)$.

Let σ be a system such that $\nu(\alpha,\sigma) \neq \perp$, let $\sigma' \in \mathscr{S}[\alpha](\sigma)$. Noting that α is a deterministic adversary, we immediately get

$$\tau(\alpha,\sigma')_n = \tau(\alpha,\sigma)_n \;\forall n < \nu(\alpha,\sigma)$$

and

$$\nu(\alpha,\sigma') = \nu(\alpha,\sigma)$$

from which immediately follows

$$\mathscr{S}[\alpha](\sigma') = \mathscr{S}[\alpha](\sigma)$$

Let α be a non-redundant adversary, let σ, σ' be systems in $\mathscr{N}(\alpha)$. Let $\sigma\, \mathscr{S}[\alpha]\, \sigma'$ denote the relationship

$$\sigma' \in \mathscr{S}[\alpha](\sigma)$$

Since $\tau(\alpha,\sigma) \sim \sigma$, we immediately have that $\mathscr{S}[\alpha]$ is reflexive. Since we established $\mathscr{S}[\alpha](\sigma') = \mathscr{S}[\alpha](\sigma)$, $\mathscr{S}[\alpha]$ is also symmetric and transitive. Therefore, $\mathscr{S}[\alpha]$ is an equivalence relation on $\mathscr{N}(\alpha)$.

Let

$$\mathscr{S}[\alpha]_1, \ldots \mathscr{S}[\alpha]_h = \frac{\mathscr{N}(\alpha)}{\mathscr{S}[\alpha]}$$

intuitively, each $\mathscr{S}[\alpha]_i$ is a distinct set of systems that are indistinguishable to α, before the first invocation of $Sample(\ldots)$.

Let $i \in 1..h$. Let $\sigma \in \mathscr{S}[\alpha]_i$, let $E \in \mathscr{E}[\alpha](\sigma)$, let σ' be identical to σ, with the exception of $\pi(\alpha,\sigma)$'s echo sample for $m(\alpha,\sigma)$, which is replaced with E. As we previously proved, $\tau(\alpha,\sigma)_{1..(\nu(\alpha,\sigma)-1)} \sim \sigma'$, therefore have $\sigma' \in \mathscr{S}[\alpha]_i$. Moreover, we proved that for every σ in $\mathscr{S}[\alpha]_i$, $\mathscr{E}[\alpha](\sigma)$ yields the same set of samples.

Let σ, σ' be systems in $\mathscr{S}[\alpha]_i$, let $\pi^* = \pi(\alpha,\sigma) = \pi(\alpha,\sigma')$, let $m^* = m(\alpha,\sigma) = m(\alpha,\sigma')$. Let $\sigma\, \mathscr{E}[\alpha]\, \sigma'$ denote the relationship

$$\sigma[\pi^*][m^*] = \sigma'[\pi^*][m^*] \in (\mathscr{E}[\alpha](\sigma) = \mathscr{E}[\alpha](\sigma'))$$

from its definition we can immediately see that $\mathscr{E}[\alpha]$ is an equivalence relation, and we can partition

$$\mathscr{E}[\alpha]^i_1, \ldots, \mathscr{E}[\alpha]^i_l = \frac{\mathscr{S}[\alpha]_i}{\mathscr{E}[\alpha]}$$

with

$$\left|\mathscr{E}[\alpha]^i_1\right| = \ldots = \left|\mathscr{E}[\alpha]^i_l\right|$$

Let $\mathscr{C}[\alpha]^i_1, \ldots, \mathscr{C}[\alpha]^i_l$ denote the probability of α compromising a random element of $\mathscr{E}[\alpha]^i_1, \ldots, \mathscr{E}[\alpha]^i_l$:

$$\mathscr{C}[\alpha]^i_j = \frac{\left|\left\{\sigma \in \mathscr{E}[\alpha]^i_j \mid \alpha \searrow \sigma\right\}\right|}{\left|\mathscr{E}[\alpha]^i_j\right|}$$

we can determine the subset whose consistency α has the highest probability of compromising by

$$\mathscr{C}[\alpha]^i_* = \arg\max_j \mathscr{C}[\alpha]^i_j$$

Finally, we define an **optimization function** $\mathscr{O}[\alpha] : \mathscr{N}(\alpha) \to \mathscr{N}(\alpha)$. Let $\sigma \in \mathscr{S}[\alpha]_i$, we define $\mathscr{O}[\alpha]$ by

$$\mathscr{O}[\alpha](\sigma)[\pi][m] = \begin{cases} \mathscr{E}[\alpha](\sigma)_{\mathscr{C}[\alpha]^i_*} & \text{iff } \pi = \pi(\alpha,\sigma), m = m(\alpha,\sigma) \\ \sigma[\pi][m] & \text{otherwise} \end{cases}$$

As we previously proved, every $\mathscr{E}[\alpha]^i_j$ has the same number of elements. Moreover, $\mathscr{O}[\alpha]$ maps a system σ in $\mathscr{E}[\alpha]^i_j$ to the corresponding system σ' in $\mathscr{E}[\alpha]^i_{C[\alpha]^i_*}$ that is identical to σ, except for $\pi(\alpha,\sigma)$'s echo sample for $m(\alpha,\sigma)$, which is replaced with $\mathscr{E}[\alpha](\sigma)_{\mathscr{C}[\alpha]^i_*}$.

Therefore, for every $\sigma, \sigma' \in \mathscr{E}^i_{\mathscr{C}[\alpha]^i_*}$,

$$\left|\mathscr{O}[\alpha]^{-1}(\sigma)\right| = \left|\mathscr{O}[\alpha]^{-1}(\sigma')\right| = \frac{|\mathscr{S}[\alpha]_i|}{\left|\mathscr{E}[\alpha]^i_1\right|}$$

System masking. Let α be a non-redundant adversary, let $\alpha' = \Delta_{sb}(\alpha)$, let σ be a system.

We start by noting that, if $\nu(\alpha,\sigma) = \bot$, then $\tau(\alpha,\sigma) = \tau(\alpha',\sigma)$. Indeed, if α never invokes $Sample(\ldots)$ when coupled with σ, all calls to $smadv$ are simply forwarded to the corresponding calls in sys. Therefore, if α compromises the consistency of σ, then trivially α' compromises the consistency of σ as well.

Let us assume that $\nu(\alpha,\sigma) \neq \bot$. Let σ' be an identical copy of σ. We start by noting that, if we couple Sample-masking decorator with σ', we effectively obtain a system instance δ

with which α directly exchanges invocations and responses. Here we show that the trace $\tau(\alpha, \mathcal{O}[\alpha](\sigma))$ is identical to the trace $\tau(\alpha, \delta)$. Intuitively, this means that α has no way of *distinguishing* whether it has been coupled directly with $\mathcal{O}[\alpha](\sigma)$, or it has been coupled with σ', with Non-redundant decorator acting as an interface.

We previously proved that the trace exchanged between $nradv$ and $smadv$ is identical to the trace that $nradv$ would exchange with sys, if $\pi(\alpha, \sigma)$'s echo sample for $m(\alpha, \sigma)$ in sys was replaced with $cache$.

Let $i \in \mathbb{N}$ such that $\sigma \in \mathcal{S}[\alpha]_i$. Procedure $optimize$ explicitly loops over all possible values of $sample \in \Pi^E$. For every value of $sample$, if loops over all the systems $\bar{\sigma}$ that are compatible with $trace$, and satisfy $\bar{\sigma}[\pi(\alpha, \sigma)][m(\alpha, \sigma)] = sample$. If, at the end of the loop, $systems \neq 0$, then $compromissions$ effectively represents, for some j, the number of systems in $\mathcal{E}[\alpha]^i_j$ that α compromises. Since $optimize$ selects the value of $sample$ that maximizes $compromissions / system$, the value that is eventually assigned to $cache$ is effectively $\mathcal{E}[\alpha](\sigma)_{\mathcal{C}[\alpha]^i_*}$, which proves the statement.

We previously proved that, if α compromises the consistency of $\mathcal{O}[\alpha](\sigma)$, then $\tau(\alpha, \mathcal{O}[\alpha](\sigma))$ compromises the consistency of σ as well. Noting that every invocation to $smadv.Deliver(\ldots)$ or $smadv.Echo(\ldots)$ is respectively forwarded to $sys.Deliver(\ldots)$ or $sys.Echo(\ldots)$, we finally obtain that if α compromises the consistency of $\mathcal{O}[\alpha](\sigma)$, then α' compromises the consistency of σ as well.

Adversarial power. We can finally show that the adversarial power of α' is greater than the adversarial power of α. Let σ be a system.

As we previously established, if $\sigma \notin \mathcal{N}(\alpha)$, then the probability of α compromising σ is identical to the probability of α compromising σ'.

Let us assume that $\sigma \in \mathcal{N}(\alpha)$. Let $i, j \in \mathbb{N}$ such that $\sigma \in \mathcal{E}[\alpha]^i_j$. The probability of α compromising the consistency of σ is

$$\mathcal{P}[\alpha \searrow \sigma] = \mathcal{C}[\alpha]^i_j$$

and, since α' compromises the consistency of σ if α compromises the consistency of $\mathcal{O}[\alpha](\sigma)$, the probability of α' compromising the consistency of σ is

$$\mathcal{P}\left[\alpha' \searrow \sigma\right] = \mathcal{P}[\alpha \searrow \mathcal{O}[\alpha](\sigma)] = \mathcal{C}[\alpha]^i_{\mathcal{C}[\alpha]^i_*} \geq \mathcal{C}[\alpha]^i_j = \mathcal{P}[\alpha \searrow \sigma]$$

Which proves that the adversarial power of α' is greater or equal to the adversarial power of α. □

3.11.6 Byzantine-counting adversary

Lemma 25. *The set of Byzantine-counting adversaries $\mathscr{A}_{bc}$ is optimal.*

Proof. We again prove the result using a decorator. Here we show that a decorator Δ_{bc} exists such that, for every $\alpha \in \mathscr{A}_{sb}$, the adversary $\alpha' = \Delta_{bc}(\alpha)$ is a Byzantine-counting adversary, and more powerful than α. If this is true, then the lemma is proved: let α^* be an optimal adversary, then the Byzantine-counting $\alpha^+ = \Delta_{bc}(\alpha^*)$ is optimal as well.

Decorator. Algorithm 12 implements **Byzantine-counting decorator**, a decorator that transforms a sample-blind adversary into a Byzantine-counting adversary. Provided with a sample-blind adversary $sbadv$, Byzantine-counting decorator acts as an interface between $sbadv$ and a system sys, effectively implementing a Byzantine-counting adversary $bcadv$. Byzantine-counting decorator exposes both the adversary and the system interface: the underlying adversary $sbadv$ uses $bcadv$ as its system.

Byzantine-counting decorator works as follows:

- Procedure $bcadv.Init()$ generates $best.byzantine$, an array of C pre-computed responses that $bcadv$ will provide to any subsequent invocation of $bcadv.Byzantine(\ldots)$, optimized to maximize the probability of compromising sys. This is achieved as follows:
  - The procedure loops over every correct process π, and queries $\left|sys.Byzantine(\pi)\right|$ to determine how many Byzantine processes there are in the first echo sample of π. For each π, the procedure sets variable $space[\pi]$ to the set of all possible responses to $bcadv.Byzantine(\pi)$ that satisfy the condition $\left|bcadv.Byzantine(\pi)\right| = \left|sys.Byzantine(\pi)\right|$.
  - The procedure loops over every possible array $byzantine$ of C responses that, for every $\pi \in \Pi_C$, satisfies $byzantine[\pi] \in space[\pi]$. It then counts the number of systems σ that are compatible with $byzantine$ (i.e., that satisfy, for every $\pi \in \Pi_C$, $\sigma.Byzantine(\pi) = byzantine[\pi]$) and whose consistency is compromised by the underlying adversary $SampleBlindAdversary$.
  - The procedure sets $best.byzantine$ to the array $byzantine$ that maximizes the number of systems compatible with $byzantine$ whose consistency is compromised by $SampleBlindAdversary$.
- Procedure $bcadv.Byzantine(process)$ simply returns $best.byzantine[process]$.
- Procedure $bcadv.State()$ simply forwards the call to $sys.State()$.
- Procedure $bcadv.Sample(process, message)$ is never called. This is due to the fact that $sbadv$ is sample-blind.

Algorithm 12 Byzantine-counting decorator.

```
1:  Implements:
2:      ByzantineCountingAdversary + CobSystem, instance bcadv
3:
4:  Uses:
5:      SampleBlindAdversary, instance sbadv, system bcadv
6:      CobSystem, instance sys
7:
8:  procedure bcadv.Init() is
9:      best.byzantine = ⊥;     best.compromissions = 0;
10:     space = {⊥}^C;
11:
12:     for all π ∈ Π_C do
13:         count = |sys.Byzantine(π)|;
14:         space[π] = (Π \ Π_C)^count;
15:     end for
16:
17:     for all byzantine ∈ space[π_1] × ... × space[π_C] do
18:         compromissions = 0;
19:         for all σ ∈ 𝒮 do
20:             match = True;
21:             for all π ∈ Π_C do
22:                 if σ.Byzantine(π) ≠ byzantine[π] then
23:                     match ← False;
24:                 end if
25:             end for
26:
27:             if match and SampleBlindAdversary ↘ σ then
28:                 compromissions ← compromissions + 1;
29:             end if
30:         end for
31:
32:         if compromissions > best.compromissions then
33:             best.byzantine ← byzantine;
34:             best.compromissions = compromissions;
35:         end if
36:     end for
37:     sbadv.Init();
38:
39: procedure bcadv.Step() is
40:     sbadv.Step();
41:
42: procedure bcadv.Byzantine(process) is
43:     return best.byzantine[process];
44:
```

```
45: procedure bcadv.State() is
46:     return sys.State();
47:
48: procedure bcadv.Sample(process, message) is
49:     raise error;
50:
51: procedure bcadv.Deliver(process, message) is
52:     sys.Deliver(process, message);
53:
54: procedure bcadv.Echo(process, sample, source, message) is
55:     sys.Echo(process, sample, source, message);
56:
57: procedure bcadv.End() is
58:     sys.End();
59:
```

- Procedure $bcadv.Deliver(process, message)$ simply forwards the call to $sys.Deliver(process, message)$.
- Procedure $bcadv.Echo(process, sample, source, message)$ simply forwards the call to $sys.Echo(process, sample, source, message)$.
- Procedure $bcadv.End()$ simply forwards the call to $sys.End()$.

Correctness. We start by proving that no adversary has undefined behavior when coupled with `Byzantine-counting decorator`. An adversary has undefined behavior if, at any point, the sequence of invocations and responses it exchanges with $bcadv$ is incompatible with every system.

Upon initialization, $bcadv$ generates an array $best.byzantine$ of C responses, one for every call to $bcadv.Byzantine(\pi \in \Pi_C)$. For every correct process π, $best.byzantine[\pi]$ contains only Byzantine processes and satisfies $\left|best.byzantine[\pi]\right| = \left|sys.Byzantine(\pi)\right|$. Let sys' be the system obtained by replacing the Byzantine component of each correct process π's echo samples in sys with $best.byzantine[\pi]$. The trace exchanged between $sbadv$ and $bcadv$ is always compatible with sys'. Indeed:

- Every call to $bcadv.Byzantine(\pi)$ returns $best.byzantine[\pi]$ which is equal, by definition, to $sys.Byzantine(\pi)$.
- Every call to $bcadv.State()$ is simply forwarded to $sys.State()$. Let π be a correct process, let m be a message. Since that $bcadv$ is an auto-echo adversary, when $bcadv.State()$ is invoked, every Byzantine process in π's echo sample for m has sent an `Echo(`m`, `m`)` message both in sys and sys'. Moreover, the number of Byzantine processes in π's echo sample for m is identical in sys and sys'. Finally, set of cor-

rect processes in π's echo sample for m is identical in sys and sys'. Consequently, $bcadv.State() = sys.State() = sys'.State()$.

Byzantine-counting. It is immediate to see that Byzantine-counting decorator always implements a Byzantine-counting adversary. Indeed, for any $\pi \in \Pi_C$, $sys.Byzantine(\pi)$ is only invoked from $|sys.Byzantine(\pi)|$.

Byzantine interchangeability. Let α be a sample-blind system. Let σ be a system, let σ' be a system such that, for every correct process π, every message m, and every $n \in 1..E$,

$$(\sigma[\pi][m][n] \in \Pi_C) \implies (\sigma'[\pi][m][n] = \sigma[\pi][m][n])$$
$$(\sigma[\pi][m][n] \notin \Pi_C) \implies (\sigma'[\pi][m][n] \notin \Pi_C)$$

In other words, for every $\pi \in \Pi_C$ and every $m \in \mathcal{M}$, the set of correct processes in π's echo sample for m is identical in σ and σ'.

Here we prove that, if α compromises σ, then $\tau(\alpha, \sigma)$ compromises σ'. In order to do this, we first establish some auxiliary results.

Let us consider the case where α is run against σ and $\tau(\alpha, \sigma)$ is applied to σ'. Let π be a correct process, let ρ be a process, let m be a message. At the end of both adversarial executions, the following hold true:

- If π pb.Delivered m in σ, then π pb.Delivered m in σ' as well. This follows immediately from the fact that $\tau(\alpha, \sigma)$ is applied to σ', and $((\texttt{Deliver}, \pi, m), \perp) \in \tau(\alpha, \sigma)$.

- If ρ sent an `Echo`(m, m) message to π in σ, then ρ sent an `Echo`(m, m) message to π in σ'. Indeed, if ρ is a correct process, and it sent an `Echo`(m, m) message to π in σ, then it pb.Delivered m both in σ and σ'. Therefore, it sent an `Echo`(m, m) message to π in σ' as well. If ρ is a Byzantine process then, since α is an auto-echo adversary, ρ sent an `Echo`(m, m) message to π both in σ and σ'.

- If π delivered m in σ, then π also delivered m in σ'. This follows from the above, and the fact that the correct processes in π's echo sample for m are identical in σ and σ'.

If α compromises the consistency of σ, then two correct processes π, π' and two distinct messages m, $m' \neq m$ exist such that π delivered m, and π' delivered m' in σ. From the above, however, π delivered m, and π' delivered m', in σ' as well. Consequently, $\tau(\alpha, \sigma)$ compromises the consistency of σ'.

System optimization. Let σ, σ' be systems. We define the relationship $\overset{|F|}{\sim}$ by

$$\left(\sigma \overset{|F|}{\sim} \sigma'\right) \overset{def}{\Longleftrightarrow} \left(\forall \pi \in \Pi_C, \forall n \in 1..E, \sigma[\pi][1][n] \in \Pi_C \Leftrightarrow \sigma'[\pi][1][n] \in \Pi_C\right)$$

In other words, $\sigma \overset{|F|}{\sim} \sigma'$ if, for every π and for every $n \in 1..E$, the n-th element of the first of π's echo samples is either correct both in σ and σ', or Byzantine both in σ and σ'.

It is immediate to see that $\overset{|F|}{\sim}$ is an equivalence relation. We can therefore partition $\mathscr{S}$ with $\overset{|F|}{\sim}$ to obtain

$$\mathscr{S}_1, \ldots, \mathscr{S}_h = \frac{\mathscr{S}}{\overset{|F|}{\sim}}$$

Let σ, σ' be systems. We define the relationship $\overset{F}{\sim}$ by

$$\left(\sigma \overset{F}{\sim} \sigma'\right) \overset{def}{\Longleftrightarrow} \left(\forall \pi \in \Pi_C, \forall n \in 1..E, \sigma[\pi][1][n] \notin \Pi_C \Leftrightarrow \sigma'[\pi][1][n] = \sigma[\pi][1][n]\right)$$

Intuitively, $\sigma \overset{F}{\sim} \sigma'$ if the Byzantine processes in each echo sample are identical in σ and σ'. Again, $\overset{F}{\sim}$ is an equivalence relation that we can use to partition $\mathscr{S}_i$:

$$\mathscr{S}^i_1, \ldots, \mathscr{S}^i_l = \frac{\mathscr{S}_i}{\overset{F}{\sim}}$$

and noting that, in Simplified Sieve, every correct process selects independently the correct processes in its echo samples, we have

$$\left|\mathscr{S}^i_1\right| = \ldots = \left|\mathscr{S}^i_l\right|$$

Let α be a sample-blind adversary. We define $\mathscr{C}[\alpha]^i_j$ as the fraction of systems in $\mathscr{S}^i_j$ whose consistency is compromised by α:

$$\mathscr{C}[\alpha]^i_j = \frac{\left|\left\{\sigma \in \mathscr{S}^i_j \mid \alpha \searrow \sigma\right\}\right|}{\left|\mathscr{S}^i_j\right|}$$

From $\mathscr{C}[\alpha]^i_j$ we can define

$$\mathscr{C}[\alpha]^i_* = \arg\max_j \mathscr{C}[\alpha]^i_j$$

Intuitively, $\mathscr{C}[\alpha]^i_*$ identifies the partition of $\mathscr{S}_i$ that α has the highest probability of compromising consistency.

Finally, we define an **optimization function** $\mathcal{O}[\alpha] : \mathcal{S} \to \mathcal{S}$. Let $\sigma \in \mathcal{S}_i$, we define $\mathcal{O}[\alpha]$ by

$$\begin{aligned} \mathcal{O}[\alpha](\sigma) &\in \mathcal{S}^i_{\mathcal{C}[\alpha]^i_*} \\ \sigma[\pi][m][n] \in \Pi_C &\implies \mathcal{O}[\alpha](\sigma)[\pi][m][n] = \sigma[\pi][m][n] \end{aligned}$$

As we previously proved, every $\mathcal{S}^i_j$ has the same number of elements. Moreover, $\mathcal{O}[\alpha]$ maps a system σ in $\mathcal{S}^i_j$ to the corresponding system σ' in $\mathcal{S}^i_{\mathcal{C}[\alpha]^i_*}$ such that every correct process in an echo sample in σ is identical to the corresponding process in σ'.

Therefore, for every $\sigma, \sigma' \in \mathcal{S}^i_{C[\alpha]^i_*}$,

$$\left|\mathcal{O}[\alpha]^{-1}(\sigma)\right| = \left|\mathcal{O}[\alpha]^{-1}(\sigma')\right| = \frac{|\mathcal{S}_i|}{\left|\mathcal{S}^i_1\right|}$$

System masking. Let α be a sample-blind adversary, let $\alpha' = \Delta_{bc}(\alpha)$, let σ be a system, let σ' be an identical copy of σ. We start by noting that, if we couple Byzantine-counting decorator with σ', we effectively obtain a system instance δ with which α directly exchanges invocations and responses. Here we show that the trace $\tau(\alpha, \mathcal{O}[\alpha](\sigma))$ is identical to the trace $\tau(\alpha, \delta)$. Intuitively, this means that α has no way of *distinguishing* whether it has been coupled directly with $\mathcal{O}[\alpha](\sigma)$, or it has been coupled with σ', with Byzantine-counting decorator acting as an interface.

We previously proved that the trace exchanged between $sbadv$ and $bcadv$ is identical to the trace that $sbadv$ would exchange with the system sys' that is obtained by replacing the Byzantine component of each correct process π's echo samples in sys with $best.byzantine[\pi]$.

Let $i \in \mathbb{N}$ such that $\sigma \in \mathcal{S}_i$. Procedure $bcadv.Init()$ explicitly loops over all the possible values of $byzantine$ that satisfy the condition $\left|byzantine[\pi]\right| = \left|sys.Byzantine(\pi)\right|$ for all $\pi \in \Pi_C$. It then loops over every system σ that satisfies $\sigma.Byzantine(\pi) = byzantine[\pi]$, and counts the number of systems that α compromises. It finally selects the value of $byzantine$ that maximizes the number of compromissions. In doing so, $bcadv.Init()$ is effectively looping over every $\mathcal{S}^i_j$, and selecting the j that maximizes the probability of α compromising a random element of $\mathcal{S}^i_j$. Since $bcadv.Init()$ is effectively masking σ with the element of $\mathcal{S}^i_{\mathcal{C}[\alpha]^i_*}$ with which σ shares the correct component of every sample, the trace $\tau(\alpha, \mathcal{O}[\alpha](\sigma))$ is identical to the trace $\tau(\alpha, \delta)$.

We previously proved that, if α compromises the consistency of $\mathcal{O}[\alpha](\sigma)$, then $\tau(\alpha, \mathcal{O}[\alpha](\sigma))$ compromises the consistency of σ as well. Noting that every invocation to $bcadv.Deliver(\ldots)$ or $bcadv.Echo(\ldots)$ is respectively forwarded to $sys.Deliver(\ldots)$ or $sys.Echo(\ldots)$, we finally obtain that if α compromises the consistency of $\mathcal{O}[\alpha](\sigma)$, then α' compromises the consistency of σ as well.

Adversarial power. We can finally show that the adversarial power of α' is greater than the adversarial power of α. Let σ be a system.

Let $i, j \in \mathbb{N}$ such that $\sigma \in \mathscr{S}_j^i$. The probability of α compromising the consistency of σ is

$$\mathscr{P}[\alpha \searrow \sigma] = \mathscr{C}[\alpha]_j^i$$

and, since α' compromises the consistency of σ if α compromises the consistency of $\mathscr{O}[\alpha](\sigma)$, the probability of α' compromising the consistency of σ is

$$\mathscr{P}\left[\alpha' \searrow \sigma\right] = \mathscr{P}[\alpha \searrow \mathscr{O}[\alpha](\sigma)] = \mathscr{C}[\alpha]^i_{\mathscr{C}[\alpha]^i_*} \geq \mathscr{C}[\alpha]_j^i = \mathscr{P}[\alpha \searrow \sigma]$$

Which proves that the adversarial power of α' is greater or equal to the adversarial power of α. □

3.11.7 Single-response adversary

Lemma 26. *The set of single-response adversaries $\mathscr{A}_{sr}$ is optimal.*

Proof. We again prove the result using a decorator. Here we show that a decorator Δ_{sr} exists such that, for every $\alpha \in \mathscr{A}_{bc}$, the adversary $\alpha' = \Delta_{sr}(\alpha)$ is a single-response adversary, and as powerful as α. If this is true, then the lemma is proved: let α^* be an optimal adversary, then the sequential $\alpha^+ = \Delta_{sr}(\alpha^*)$ is optimal as well.

Decorator. Algorithm 13 implements **Single-response decorator**, a decorator that transforms a Byzantine-counting adversary into a single-response adversary. Provided with a Byzantine-counting adversary $bcadv$, Single-response decorator acts as an interface between $bcadv$ and a system sys, effectively implementing a single-response adversary $sradv$. Single-response decorator exposes both the adversary and the system interfaces: the underlying adversary $bcadv$ uses $sradv$ as its system.

Single-response decorator works as follows:

- Procedure $sradv.Init()$ initializes the following variables:
  - A $cache$ set, initially empty: $cache$ is used to store the first non-empty set returned from $sys.State()$.
  - A $poisoned$ variable: $poisoned$ is set to `True` if and only if at least one correct process in sys is poisoned. This condition is verified by looping over $sys.Byzantine(\pi)$ for every correct process π.

Algorithm 13 Single-response decorator.

```
1: Implements:
2:     SingleResponseAdversary + CobSystem, instance sradv
3:
4: Uses:
5:     ByzantineCountingAdversary, instance bcadv, system sradv
6:     CobSystem, instance sys
7:
8: procedure sradv.Init() is
9:     cache = ∅;          poisoned = False;      step = 0;
10:
11:    for all π ∈ Π_C do
12:        if |sys.Byzantine(π)| ≥ Ê then
13:            poisoned ← True;
14:        end if
15:    end for
16:
17:    bcadv.Init();
18:
19: procedure sradv.Step() is
20:    step ← step + 1;
21:
22:    if poisoned = False or step ≤ (N − C)C^2 then
23:        bcadv.Step();
24:    else if step ≤ (N − C)C^2 + C then
25:        sys.Deliver(ζ(step − (N − C)C^2), 1);
26:    else
27:        sys.End();
28:    end if
29:
30: procedure sradv.Byzantine(π) is
31:    count = |sys.Byzantine(π)|;
32:    return {⊥}^count;
33:
34: procedure sradv.State() is
35:    return cache;
36:
37: procedure sradv.Sample(process, message) is
38:    raise error;
39:
```

```
40: procedure sradv.Deliver(process, message) is
41:     sys.Deliver(process, message);
42:
43:     if cache = ∅ then
44:         cache ← sys.State();
45:     end if
46:
47: procedure sradv.Echo(process, sample, source, message) is
48:     sys.Echo(process, sample, source, message);
49:
50: procedure sradv.End() is
51:     sys.End();
52:
```

  - A $step$ variable, initially set to 0: at any time, $step$ counts how many times $sradv.Step()$ has been invoked.

- Procedure $sradv.Step()$ increments $step$, then implements two different behaviors depending on the value of $poisoned$:
  - If $poisoned$ = `True`, it forwards the call to $bcadv.Step()$ for the first $(N-C)C^2$ times. For the next C steps, it sequentially invokes $sys.Deliver(\zeta(1), 1)$, ..., $sys.Deliver(\zeta(C), 1)$. Finally, it calls $sys.End()$.
  - If $poisoned$ = `False`, it forwards the call to $bcadv.Step()$.

- Procedure $sradv.Byzantine(process)$ returns an array of $count$ elements, $count$ being the number of elements returned by $sys.Byzantine(process)$. The array is filled with $\perp$ values: since $bcadv$ is Byzantine-counting, the content of the array is irrelevant.

- Procedure $sradv.State()$ simply returns $cache$.

- Procedure $sradv.Sample(process, message)$ is never called. This is due to the fact that $bcadv$ is sample-blind.

- Procedure $sradv.Deliver(process, message)$ forwards the call to $sys.Deliver(process, message)$. Then, if $cache$ is empty, it updates $cache$ with $sys.State()$.

- Procedure $sradv.Echo(process, sample, source, message)$ simply forwards the call to $sys.Echo(process, sample, source, message)$.

- Procedure $sradv.End()$ simply forwards the call to $sys.End()$.

Correctness. Here we prove that every adversary, when coupled with `Single-response adversary`:

- Has a well-defined behavior. An adversary has undefined behavior if, at any point, the sequence of invocations and responses it exchanges with $sradv$ is incompatible with every system.

- Is process-sequential, sequential, and Byzantine-counting.

We start by noting that $poisoned = \texttt{True}$ if and only if sys is poisoned. Indeed, $sradv.Init()$ explicitly checks if any correct process has at least $\hat{E}$ Byzantine processes in its first echo sample.

We distinguish two cases, based on the value of $poisoned$. Let us assume $poisoned = \texttt{True}$. When $sradv.Step()$ is invoked, the call is forwarded to $bcadv.Step()$ only for the first $(N-C)C^2$ times. Noting that $bcadv$ is an auto-echo adversary, every call to $bcadv.Step()$ results in a call to $sradv.Echo(\ldots)$. For the next C steps, $sradv.Step()$ sequentially causes $\zeta(1), \zeta(2), \ldots$ to pb.Deliver message 1. Finally, $sradv.Step()$ invokes $sys.End()$. Therefore, $sradv$ has a well defined behavior and implements a process-sequential adversary. Since it causes only message 1 to be pb.Delivered, $sradv$ is also trivially sequential.

Let us assume $poisoned = \texttt{False}$. As we proved in Section 3.9, since sys is not poisoned, a correct process in sys will only deliver a message m as a result of an invocation to $sys.Deliver(\pi, m)$ for some $\pi \in \Pi_C$. Until $cache \neq \emptyset$, $cache$ is updated to $sys.State()$ after every call to $sys.Deliver(\ldots)$. Therefore, throughout the first phase, $sradv.State()$ is always identical to $sys.State()$. The trace exchanged between $bcadv$ and $sradv$ is, therefore, trivially compatible with sys.

Throughout the second phase, we have $cache \neq \bot$. Since, throughout the first phase, $cache$ is updated after every call to $sys.Deliver(\ldots)$, only one message m^* exists such that, for some $\pi^* \in \Pi_C$, $(\pi^*, m^*) \in cache$. Noting that $bcadv$ is a non-redundant adversary, it will never invoke $sradv.Deliver(\ldots)$ on m^*: indeed, the value returned from $sradv.State()$ never changes throughout the second phase. We define a system sys' by

$$sys'[\pi][m][n] = \begin{cases} sys[\pi][m][n] & \text{iff } m = m^* \text{ or } sys[\pi][m][n] \in \Pi \setminus \Pi_C \\ \pi^* & \text{otherwise} \end{cases}$$

The trace exchanged between $bcadv$ and $sradv$ is compatible with sys'. Indeed, for every $\pi \in \Pi_C$, π's sample for m^* in sys is identical to π's echo sample for m^* in sys': at any moment, π delivered m^* in sys if and only if π delivered m^* in sys'. For every $\pi \in \Pi_C$ and $m \neq m^* \in \mathcal{M}$, π's every correct process in π's sample for m is π^*. However, π^* pb.Delivered $m^* \neq m$. Therefore, since sys' is not poisoned, no correct process in sys' ever delivers a message other than m^*.

Every call to $sradv.Deliver(\ldots)$ and $sradv.Echo(\ldots)$ is respectively forwarded to $sys.Deliver(\ldots)$ and $sys.Echo(\ldots)$. Moreover, $bcadv$ is process-sequential and sequential. Therefore, if $poisoned = \texttt{True}$, $sradv$ is also process-sequential and sequential.

It is immediate to see that Single-response decorator always implements a Byzantine-counting adversary. Indeed, for any $\pi \in \Pi_C$, $sys.Byzantine(\pi)$ is only invoked from $\left|sys.Byzantine(\pi)\right|$.

Single-response. It is immediate to see that Single-response decorator always implements a single-response adversary. Indeed, when $sys.State()$ returns a non-empty set for the first time, $cache$ is set to a non-empty set, and $sys.State()$ is never invoked again.

Roadmap. Let $\alpha \in \mathscr{A}_{bc}$, let $\alpha' = \Delta_{sr}(\alpha)$. Let σ be a system such that α compromises the consistency of σ. Let σ' be an identical copy of σ. In order to prove that α' is as powerful as α, we prove that α' compromises the consistency of σ'.

Poisoned case. Noting that α' is an auto-echo adversary, if σ is poisoned we immediately have that α' compromises the consistency of σ'.

Trace. Let us assume that σ is not poisoned. We start by noting that, if we couple Single-response decorator with σ', we effectively obtain a system instance δ with which α directly exchanges invocations and responses.

We start by defining a boolean sequence W by setting $W_n = \texttt{True}$ if and only if, after the n-th invocation, two correct processes π, π' and two distinct messages $m, m' \neq m$ exist such that π delivered m and π' delivered m' in σ. Since α compromises the consistency of σ, for some n we have $W_n = \texttt{True}$. Let

$$w = \min n \mid W_n = \texttt{True}$$

Here we show that, for every $n \leq w$, the trace $\tau(\alpha,\sigma)_n$ is identical to the trace $\tau(\alpha,\delta)_n$. Intuitively, this means that, until the consistency of σ is compromised, α has no way of *distinguishing* whether it has been coupled directly with σ, or it has been coupled with σ', with Single-response decorator acting as an interface. We prove this by induction.

Let us assume

$$\begin{aligned} \tau(\alpha,\sigma) &= ((i_1, r_1), \ldots) \\ \tau(\alpha,\delta) &= ((i'_1, r'_1), \ldots) \\ i_j = i'_j, r_j = r'_j & \qquad \forall j \leq n \end{aligned}$$

We start by noting that, since α is a deterministic algorithm, we immediately have

$$i_{n+1} = i'_{n+1}$$

and we need to prove that $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{Byzantine}, \pi)$. Since procedure $sradv.Byzantine(\pi)$ forwards the call to $sys.Byzantine(\pi)$, $bcadv$ is a Byzantine-counting adversary, and σ' is an identical copy of σ, with a minor abuse of notation we effectively have $r_{n+1} = r'_{n+1}$.

Let us assume that $i_{n+1} = (\texttt{State})$. We start by noting that, since all calls to $sradv.Deliver(\ldots)$ and $sradv.Echo(\ldots)$ are respectively forwarded to $sys.Deliver(\ldots)$ and $sys.Echo(\ldots)$, a correct process π delivered m^* in σ if and only if it delivered m^* in σ' as well. As we proved, throughout the first phase, $sradv.State()$ always returns the same value as $sys.State()$. Let us assume $n > |\eta(\alpha, \sigma)|$. Let m^* be the only message that was delivered by at least one correct process in σ. Noting that a correct process delivers a message only as a result of a call to $sys.Deliver(\ldots)$, we have $n < w$. Therefore, by definition, no correct process in σ delivered a message other than m^*. Since α is a non-redundant adversary, it never causes any correct process to pb.Deliver m^* throughout the second phase. As a result, no correct process delivers m^* in σ throughout the second phase. Therefore, all the processes that delivered m^* in σ are represented in $cache$, and no other process delivered a message $m \neq m^*$. Consequently, $r_{n+1} = r'_{n+1}$.

Noting that procedures $Deliver(\ldots)$ and $Echo(\ldots)$ never return a value, we trivially have that if $i_{n+1} = (\texttt{Deliver}, \pi, m)$ or $i_{n+1} = (\texttt{Echo}, \pi, s, \xi, m)$ then $r_{n+1} = \perp = r'_{n+1}$. By induction, we have that, for every $n \leq w$, $\tau(\alpha, \sigma)_n = \tau(\alpha, \delta)_n$.

Consistency of σ'. We proved that, for all $n \leq w$, $\tau(\alpha, \sigma)_n = \tau(\alpha, \delta)_n$. Moreover, we proved that if a correct process π eventually delivers a message m in σ before the w-th invocation, then π also delivers m in σ' before the w-th invocation.

Since α compromises the consistency of σ after the w-th invocation, two correct processes π, π' and two distinct messages m, $m' \neq m$ exist such that, in σ, π delivered m and π' delivered m' before the w-th invocation. Therefore, in σ', π delivered m and π' delivered m' before the w-th invocation. Therefore α' compromises the consistency of σ'.

Consequently, the adversarial power of α is equal to the adversarial power of $\alpha' = \Delta_{sr}(a)$, and the lemma is proved.

□

3.11.8 Two-phase adversary

Lemma 27. *The set of two-phase adversaries $\mathscr{A}_{tp}$ is optimal.*

Proof. We again prove the result using a decorator. Here we show that a decorator Δ_{tp} exists such that, for every $\alpha \in \mathscr{A}_{sm}$, the adversary $\alpha' = \Delta_{tp}(\alpha)$ is a two-phase adversary, and more powerful than α. If this is true, then the lemma is proved: let α^* be an optimal adversary, then the sequential $\alpha^+ = \Delta_{tp}(\alpha^*)$ is optimal as well.

Decorator. Algorithm 14 implements **Two-phase decorator**, a decorator that transforms a state-polling adversary into a two-phase adversary. Provided with a state-polling adversary $spadv$, Two-phase decorator acts as an interface between $spadv$ and a system sys, effectively implementing a single-response adversary $tpadv$. Two-phase decorator exposes both the adversary and the system interfaces: the underlying adversary $spadv$ uses $tpadv$ as its system.

Two-phase decorator works as follows:

- Procedure $tpadv.Init()$ initializes a $invocations$ variable: at any time, $invocations$ counts the number of invocations issued by $spadv$.
- Procedure $compatible(invocations)$ returns a set of systems σ that satisfy the following properties:
  - For every correct process π, the number of Byzantine processes in π's first echo sample is identical in σ and sys.
  - The length $|\eta(\alpha, \sigma)|$ of the first phase when α is coupled with σ is equal to $invocations$.
- Procedure $tpadv.Step()$ simply forwards the call to $spadv.Step()$.
- Procedure $tpadv.Byzantine(process)$ increments $invocations$, then returns an array of $count$ elements, $count$ being the number of elements returned from $sys.Byzantine(process)$. The array is filled with $\perp$ values: since $spadv$ is Byzantine-counting, the content of the array is irrelevant.
- Procedure $tpadv.State()$ increments $invocations$. It then returns an empty set if $sys.State()$ is empty. If $sys.State()$ is not empty, the procedure returns, among all the possible responses that are compatible with the trace exchanged between $spadv$ and $tpadv$, the one that maximizes the probability of $spadv$ compromising the consistency of sys. This is achieved as follows:
  - The procedure loops over every system σ in the set $compatible(invocations)$. In doing so, the procedure loops over every system σ such that: σ has the same

Algorithm 14 Two-phase decorator.

```
1: Implements:
2:     TwoPhaseAdversary + CobSystem, instance tpadv
3:
4: Uses:
5:     StatePollingAdversary, instance spadv, system tpadv
6:     CobSystem, instance sys
7:
8: procedure tpadv.Init() is
9:     invocations = 0;
10:    spadv.Init();
11:
12: procedure compatible(invocations) is
13:    systems = ∅;
14:
15:    for all σ ∈ 𝒮 do
16:        match = True;
17:        for all π ∈ Π_C do
18:            if |σ.Byzantine(π)| ≠ |sys.Byzantine(π)| then
19:                match = False;
20:            end if
21:        end for
22:
23:        if match = True and |η(α,σ)| = invocations then
24:            systems ← systems ∪ {σ};
25:        end if
26:    end for
27:
28:    return systems;
29:
30: procedure tpadv.Step() is
31:    spadv.Step();
32:
33: procedure tpadv.Byzantine(process) is
34:    invocations ← invocations + 1;
35:    count = sys.Byzantine(process);
36:    return {⊥}^count;
37:
```

```
38: procedure tpadv.State() is
39:     invocations ← invocations + 1;
40:
41:     if sys.State() ≠ ∅ then
42:         outcomes = ∅;
43:         for all σ ∈ compatible(invocations) do
44:             (invocation, response) = τ(α, σ)_invocations;
45:             outcomes ← outcomes ∪ {(response, τ(α, σ))};
46:         end for
47:
48:         best.response = ⊥;    best.compromissions = 0;
49:
50:         for all (response, fulltrace) ∈ outcomes do
51:             compromissions = 0;
52:
53:             for all σ ∈ compatible(invocations) do
54:                 if fulltrace ↘ σ then
55:                     compromissions ← compromissions + 1;
56:                 end if
57:             end for
58:
59:             if compromissions > best.compromissions then
60:                 best.response ← response;
61:                 best.compromissions = compromissions;
62:             end if
63:         end for
64:
65:         return best.response;
66:     else
67:         return ∅;
68:     end if
69:
70: procedure tpadv.Sample(process, message) is
71:     raise error;
72:
73: procedure tpadv.Deliver(process, message) is
74:     invocations ← invocations + 1;
75:     sys.Deliver(process, message);
76:
77: procedure tpadv.Echo(process, sample, source, message) is
78:     invocations ← invocations + 1;
79:     sys.Echo(process, sample, source, message);
80:
81: procedure tpadv.End() is
82:     sys.End();
83:
```

Byzantine count as sys; when α is coupled with σ, it concludes the first phase in exactly $invocations$ invocations.

- For every process σ in $compatible(invocations)$, the procedure stores in a set $outcome$ a $(response, fulltrace)$ pair, $response$ being the $State()$ of σ at the end of the first phase ($response$ is extracted from $\tau(\alpha,\sigma)_{invocations}$), and $fulltrace$ being $\tau(\alpha,\sigma)$, the full trace exchanged between α and σ.
- For every $(response, fulltrace)$ in $outcomes$, the procedure loops over every system σ in $compatible(invocations)$, and counts the number of systems whose consistency is compromised by $fulltrace$. The procedure returns the value of $response$ that maximizes the number of systems in $compatible(invocations)$ whose consistency is compromised by $fulltrace$.

- Procedure $tpadv.Sample(process, message)$ is never called. This is due to the fact that $spadv$ is sample-blind.
- Procedure $tpadv.Deliver(process, message)$ increments $invocations$, then forwards the call to $sys.Deliver(process, message)$.
- Procedure $tpadv.Echo(process, sample, source, message)$ increments $invocations$, then forwards the call to $sys.Echo(process, sample, source, message)$.
- Procedure $tpadv.End()$ simply forwards the call to $sys.End()$.

Correctness. Here we prove that every adversary, coupled with Two-phase decorator:

- Has a well-defined behavior. An adversary has undefined behavior if, at any point, the sequence of invocations and responses it exchanges with $tpadv$ is incompatible with every system.
- Is Byzantine-counting and single-response.

We start by noting that, since $invocations$ is incremented every time $spadv$ issues an invocation, when $tpadv.State()$ is invoked and $sys.State() \neq \emptyset$ we have $invocations = |\eta(\alpha,\sigma)|$.

Every invocation of a procedure in $tpadv$ is always forwarded to the corresponding procedure in sys, except for $tpadv.State()$. Whenever $sys.State() = \emptyset$, $tpadv.State()$ returns $\emptyset$ as well. Therefore, up to the $(|\eta(\alpha, sys)| - 1)$-th invocation, the trace exchanged between $spadv$ and $tpadv$ is trivially compatible with sys.

Procedure $compatible(invocations)$ returns all systems σ such that the condition $|\sigma.Byzantine(\pi)| = |sys.Byzantine(\pi)|$ holds for all $\pi \in \Pi_C$, and $|\eta(\alpha,\sigma)| = invocations = |\eta(\sigma, sys)|$. It is immediate to see that $compatible(invocations)$ is non-empty, as it includes sys. Every system $\sigma \in compatible(invocations)$ is compatible with the first $n-1$ elements

of the trace exchanged between $spadv$ and $tpadv$. Procedure $tpadv.State()$ then returns a response $response$, such that

$$\tau(\alpha,\sigma)_{invocations} = ((\texttt{State}), response)$$

for some $\sigma \in compatible(invocations)$. Therefore, the first n elements of the trace exchanged between $spadv$ and $tpadv$ is compatible with σ. Due to Lemma 15, the entire trace exchanged between $spadv$ and $tpadv$ is compatible with σ.

It is easy to see that $tpadv$ always implements a Byzantine-counting and single-response adversary. Indeed: whenever $tpadv$ invokes $sys.Byzantine(\pi)$, it invokes $\left|sys.Byzantine(\pi)\right|$; $tpadv.State()$ returns a non-empty set if and only if $sys.State()$ returns a non-empty set, and $spadv$ is a single-response adversary.

Two-phase. It is immediate to see that Two-phase decorator always implements a two-phase adversary. Indeed, whenever $tpadv$ invokes $sys.State()$, it invokes $(sys.State() \neq \emptyset)$.

System partitioning. Let α be a state-polling adversary, let σ be a system. Let us denote with $\mathscr{S}^*$ the set of non-poisoned systems. We denote with $\overset{\alpha}{\sim}$ the two conditions $\forall \pi \in \Pi_C, \forall m \in \mathscr{M}$,

$$|\{n \in 1..E \mid \sigma[\pi][m][n] \in \Pi_C\}| = \left|\left\{n \in 1..E \mid \sigma'[\pi][m][n] \in \Pi_C\right\}\right|$$

and

$$\left|\eta(\alpha,\sigma)\right| = \left|\eta(\alpha,\sigma')\right|$$

It is immediate to see that $\overset{\alpha}{\sim}$ is an equivalence relation, and we can use $\overset{\alpha}{\sim}$ to partition $\mathscr{S}^*$:

$$\mathscr{S}[\alpha]_1, \ldots, \mathscr{S}[\alpha]_h = \frac{\mathscr{S}}{\overset{\alpha}{\sim}}$$

Let $i \in 1..h$. Due to Lemma 14, we have

$$\forall \sigma,\sigma' \in \mathscr{S}[\alpha]_i, \forall n < \left|\eta(\alpha,\sigma)\right|, \ \tau(\alpha,\sigma)_n = \tau(\alpha,\sigma')_n$$

Moreover, since σ is not poisoned, $\eta(\alpha,\sigma)$ includes at least one call to $Deliver(\ldots)$. Therefore, for every $i \in 1..h$, let $\sigma \in \mathscr{S}[\alpha]_i$, we can define a function $\delta[\alpha]_i : \mathscr{M} \times 1..(\left|\eta(\alpha,\sigma\right|)$ by

$$\pi \in \delta[\alpha]_i(m,n) \overset{def}{\iff} \exists j < n \mid \tau(\alpha,\sigma)_j = ((\texttt{Deliver},\pi),\bot)$$

Intuitively, $\delta[\alpha]_i(m,n)$ represents the set of correct processes that α causes to pb.Deliver m before the n-th invocation, when α is coupled with any $\sigma \in \mathscr{S}[\alpha]_i$.

We additionally define $\pi[\alpha]_i : \mathscr{M} \to \mathbb{P}(\Pi_C)$, $\pi^-[\alpha]_i : \mathscr{M} \to \mathbb{P}(\Pi_C)$ by, let $\sigma \in \mathscr{S}[\alpha]_i$,

$$\begin{aligned}\pi[\alpha]_i(m) &= \delta[\alpha]_i(m, \left|\eta(\alpha,\sigma)\right|)\\ \pi^-[\alpha]_i(m) &= \delta[\alpha]_i(m, \left|\eta(\alpha,\sigma)\right| - 1))\end{aligned}$$

Intuitively, $\pi[\alpha]_i(m)$ represents the set of correct processes that α causes to pb.Deliver m throughout the first phase, when α is coupled with any $\sigma \in \mathscr{S}[\alpha]_i$. Noting that α is a state-polling adversary, and σ is not poisoned, then $\pi^-[\alpha]_i(m)$ represents the set of correct processes that α causes to pb.Deliver m throughout the first phase when α is coupled with any $\sigma \in \mathscr{S}[\alpha]_i$, excluding the last invocation to $Deliver(\ldots)$ in $\eta(\alpha,\sigma)$.

Finally, we define $m(\alpha)_i$ by, let $\sigma \in \mathscr{S}[\alpha]_i$

$$\tau(\alpha,\sigma)_{|\eta(\alpha,\sigma)|} = ((\texttt{Deliver}, \pi \in \Pi_C, m), \bot)$$

Intuitively, $m(\alpha)_i$ is the last message that α causes a correct process to pb.Deliver throughout the first phase, when α is coupled with any $\sigma \in \mathscr{S}[\alpha]_i$. Noting that α is a state-polling adversary, and that σ is not poisoned, m is the only message delivered by at least one correct process at the end $\eta(\alpha,\sigma)$.

Let $\sigma, \sigma' \in \mathscr{S}[\alpha]_i$. We can prove that $\overset{\alpha}{\sim}$ can be equivalently restated as $\forall \pi \in \Pi_C, \forall m \in \mathscr{M}$

$$|\{n \in 1..E \mid \sigma[\pi][m][n] \in \Pi_C\}| = \left|\left\{n \in 1..E \mid \sigma'[\pi][m][n] \in \Pi_C\right\}\right|$$

and

$$\begin{aligned}&\nexists \quad \pi \in \Pi_C \mid\\ &\quad |\{n \in 1..E \mid \sigma[\pi][m(\alpha)_i][n] \in (\pi^-[\alpha]_i(m(\alpha)_i) \cup (\Pi \setminus \Pi_C)))\}| \geq \hat{E}\\ &\exists \quad \pi \in \Pi_C \mid\\ &\quad |\{n \in 1..E \mid \sigma[\pi][m(\alpha)_i][n] \in (\pi[\alpha]_i(m(\alpha)_i) \cup (\Pi \setminus \Pi_C)))\}| \geq \hat{E}\\ &\nexists \quad m \neq m(\alpha)_i, \pi \in \Pi_C \mid\\ &\quad |\{n \in 1..E \mid \sigma[\pi][m][n] \in (\pi[\alpha]_i(m) \cup (\Pi \setminus \Pi_C))\}| \geq E\end{aligned}$$

Indeed, we are restating the condition $\left|\eta(\alpha,\sigma)\right| = \left|\eta(\alpha,\sigma')\right|$ with the following conditions:

- No correct process has, in its echo sample for $m(\alpha)_i$, at least $\hat{E}$ processes that are either Byzantine, or pb.Deliver $m(\alpha)_i$ as a result of any invocation of $Deliver(\ldots)$ in $\eta(\alpha,\sigma)$ except the last. This encodes the condition that no correct process delivers $m(\alpha)_i$ before the last invocation of $Deliver(\ldots)$ in $\eta(\alpha,\sigma)$.
- At least one correct process has, in its echo sample for $m(\alpha)_i$, at least $\hat{E}$ processes that are either Byzantine, or pb.Deliver $m(\alpha)_i$ throughout the first phase when α is coupled with σ. This encodes the condition that at least one correct process delivers $m(\alpha)$ after

the last invocation of $Deliver(\ldots)$ in $\eta(\alpha, \sigma)$.

- No correct process has, in its echo sample for $m \neq m(\alpha)_i$, at least $\hat{E}$ processes that are either Byzantine, or pb.Deliver m throughout the first phase when α is coupled with σ. This encodes the condition that no message is delivered before $m(\alpha)_i$.

Let $\sigma, \sigma' \in \mathscr{S}[\alpha]_i$. We denote with $\overset{m}{\sim}$ the condition $\forall \pi \in \Pi_C$,

$$\sigma[\pi][m(\alpha)_i] = \sigma'[\pi][m(\alpha)_i]$$

Again, $\overset{m}{\sim}$ is an equivalence relation, and can be used to partition $\mathscr{S}[\alpha]_i$:

$$\mathscr{S}[\alpha]_1^i, \ldots \mathscr{S}[\alpha]_l^i = \frac{\mathscr{S}[\alpha]_i}{\overset{m}{\sim}}$$

Let $\bar{\sigma} \in \mathscr{S}[\alpha]_i$, let $\tau = \tau(\alpha, \bar{\sigma})$. For any $\sigma \in \mathscr{S}[\alpha]_i$, τ compromises the consistency of σ if τ causes at least one correct process to deliver a message $m' \neq m(\alpha)_i$ throughout the second phase. Since this condition is independent from the echo sample for $m(\alpha)_i$ of any correct process, we finally have that, for every $j \in 1..l$,

$$\mathscr{P}\left[\tau \searrow \left(\sigma \in \mathscr{S}[\alpha]_j^i\right)\right] = \mathscr{P}[\tau \searrow (\sigma \in \mathscr{S}[\alpha]_i)]$$

Adversarial power. Here we prove that $\alpha' = \Delta_{tp}(\alpha)$ is more powerful than α. Let $\bar{\sigma}$ denote a random system in $\mathscr{S}$.

Let us assume that $\bar{\sigma}$ is poisoned. Since both α and α' are auto-echo adversaries, both compromise $\bar{\sigma}$.

Let us assume that $\bar{\sigma}$ is not poisoned. For some i, j, we therefore have $\sigma \in \mathscr{S}[\alpha]_j^i$. When $tpadv.State()$ is invoked and $sys.State() \neq \emptyset$, the procedure returns a response $best.response$ such that the trace τ^* that α issues as a result of $best.response$ satisfies

$$\tau^* = \arg\max_{\tau} \mathscr{P}[\tau \searrow (\sigma \in \mathscr{S}[\alpha]_i)]$$

As we proved in the previous section, we therefore have

$$\mathscr{P}\left[\tau^* \searrow (\sigma \in \mathscr{S}[\alpha]_i)\right] \geq \mathscr{P}[\tau \searrow (\sigma \in \mathscr{S}[\alpha]_i)] = \mathscr{P}\left[\tau \searrow \left(\sigma \in \mathscr{S}[\alpha]_j^i\right)\right]$$

which proves that, if σ is not poisoned, then the probability of α' compromising σ is greater or equal to the probability of α compromising σ.

The adversarial power of α' is therefore greater or equal to the adversarial power of α, and the lemma is proved. □

4 Contagion

In this chapter, we present in detail the **probabilistic reliable broadcast** abstraction and discuss its properties. We then present Contagion, an algorithm that implements probabilistic reliable broadcast, and evaluate its **security** and **complexity** as a function of its **parameters**.

The probabilistic reliable broadcast abstraction allows the entire set of correct processes to agree on a single message from a potentially Byzantine designated sender. Probabilistic reliable broadcast is a strictly stronger abstraction than probabilistic consistent broadcast: in the case of a Byzantine sender, while probabilistic consistent broadcast only guarantees that every correct process that delivers a message delivers the same message (**consistency**), probabilistic reliable broadcast also guarantees that either no or every correct process delivers a message (**totality**).

4.1 Interface

The **probabilistic reliable broadcast** interface (instance prb, sender σ) exposes the following two **events**:

- **Request**: $\langle prb.\text{Broadcast} \mid m \rangle$: Broadcasts a message m to all processes. This is only used by σ.
- **Indication**: $\langle prb.\text{Deliver} \mid m \rangle$: Delivers a message m broadcast by process σ.

For any $\epsilon \in [0, 1]$, we say that probabilistic reliable broadcast is ϵ-secure if:

1. **No duplication**: No correct process delivers more than one message.
2. **Integrity**: If a correct process delivers a message m, and σ is correct, then m was previously broadcast by σ.
3. ϵ-**Validity**: If σ is correct, and σ broadcasts a message m, then σ eventually delivers m with probability at least $(1 - \epsilon)$.

4. ϵ-**Totality**: If a correct process delivers a message, then every correct process eventually delivers a message with probabiity at least $(1-\epsilon)$.

5. ϵ-**Consistency**: Every correct process that delivers a message delivers the same message with probability at least $(1-\epsilon)$.

4.2 Algorithm

Algorithm 15 implements Contagion. Let π be a correct process, let m be a message. Contagion securely distributes a single message across the system as follows:

- Initially, probabilistic consistent broadcast consistently distributes the same message to a subset of the correct processes.
- π can issue a Ready message for more than one message. π issues a Ready message m when either:
  - π receives m from probabilistic consistent broadcast, or
  - π collects enough Ready messages for m from its *ready sample*.
- π delivers m if m is the first message for which π collected enough Ready messages from its delivery sample.

A correct process collects Ready messages from two randomly selected samples, the *ready sample* of size R, and the *delivery sample* of size D. A correct process issues a Ready message for m upon collecting $\hat{R}$ Ready messages for m from its ready sample, and it delivers m upon collecting $\hat{D}$ Ready messages for m from its delivery sample. We discuss the values of the four parameters of Contagion in Sections 4.4, 4.9 and 4.10.

Sampling. Upon initialization (line 12), a correct process randomly selects a **ready sample** $\mathscr{R}$ of size R, and a **delivery sample** $\mathscr{D}$ of size D. Samples are selected with replacement by repeatedly calling Ω (Algorithm 2, line 4).

Publish-subscribe. Like Sieve, Contagion uses publish-subscribe to reduce its communication complexity. This is achieved by having each correct process send Ready messages only to its ready subscription set (lines 32 and 50), and accept Ready messages only from its ready and delivery samples (lines 40 and 43).

Consistent broadcast. The designated sender σ initially broadcasts its message using probabilistic consistent broadcast (line 27). When message m is pcb.Delivered (correctly signed by σ) (line 29), a correct process sends a Ready message for m (line 33) to all the processes in its ready subscription set.

Algorithm 15 Contagion

```
1: Implements:
2:     ProbabilisticReliableBroadcast, instance prb
3:
4: Uses:
5:     AuthenticatedPointToPointLinks, instance al
6:     ProbabilisticConsistentBroadcast, instance pcb
7:
8: Parameters:
9:     R: ready sample size          R̂: contagion threshold
10:    D: delivery sample size       D̂: delivery threshold
11:
12: upon event ⟨prb.Init⟩ do
13:     ready = ∅;    delivered = False;   R̃ = ∅;
14:
15:     ℛ = sample(ReadySubscribe, R);
16:     𝒟 = sample(ReadySubscribe, D);
17:
18:     replies.ready = {∅}^R;      replies.delivery = {∅}^D
19:
20: upon event ⟨al.Deliver | π, [ReadySubscribe]⟩ do
21:     for all (message, signature) ∈ ready do
22:         trigger ⟨al.Send | π, [Ready, message, signature]⟩;
23:     end for
24:     R̃ ← R̃ ∪ {π};
25:
26: upon event ⟨prb.Broadcast | message⟩ do                    ▷ only process σ
27:     trigger ⟨pcb.Broadcast | [Send, message, sign(message)]⟩;
28:
29: upon event ⟨pcb.Deliver | [Send, message, signature]⟩ do
30:     if verify(σ, message, signature) then
31:         ready ← ready ∪ {(message, signature)};
32:         for all ρ ∈ R̃ do
33:             trigger ⟨al.Send | ρ, [Ready, message, signature]⟩;
34:         end for
35:     end if
36:
```

```
37: upon event ⟨al.Deliver | π, [Ready, message, signature]⟩ do
38:     if verify(σ, message, signature) then
39:         reply = (message, signature);
40:         if π ∈ ℛ then
41:             replies.ready[π] ← replies.ready[π] ∪ {reply};
42:         end if
43:         if π ∈ 𝒟 then
44:             replies.delivery[π] ← replies.delivery[π] ∪ {reply}
45:         end if
46:     end if
47:
48: upon exists message such that |{ρ ∈ ℛ | (message, signature) ∈ replies.ready[ρ]}| ≥ R̂
    do
49:     ready ← ready ∪ {(message, signature)};
50:     for all ρ ∈ ℛ̃ do
51:         trigger ⟨al.Send | ρ, [Ready, message, signature]⟩;
52:     end for
53:
54: upon exists message such that |{ρ ∈ 𝒟 | (message, signature) ∈
    replies.delivery[ρ]}| ≥ D̂ and delivered = False do
55:     delivered ← True;
56:     trigger ⟨prb.Deliver | message⟩;
57:
```

Contagion. Upon collecting $\hat{R}$ `Ready` messages for a message m (line 48), a correct process sends a `Ready` message for m (line 51) to all the nodes in its ready subscription set.

Delivery. Upon collecting $\hat{D}$ `Ready` messages for a message m for the first time, (line 54), a correct process delivers m (line 56).

4.3 No duplication and integrity

We start by verifying that Contagion satisfies both **no duplication** and **integrity**.

Theorem 10. *Contagion satisfies no duplication.*

Proof. A message is delivered (line 56) only if the variable $delivered$ is equal to `False` (line 54). Before any message is delivered, $delivered$ is set to `True`. Therefore no more than one message is ever delivered. □

Theorem 11. *Contagion satisfies integrity.*

Proof. Upon receiving a `Ready` message, a correct process checks its signature against the

public key of the designated sender σ (line 38), and the $(message, signature)$ pair is added to the $replies.delivery$ variable only if this check succeeds. Moreover, a message is delivered only if it is represented at least $\hat{D}$ times in $replies.delivery$ (line 54).

If σ is correct, it only signs $message$ when broadcasting (line 27). Since we assume that cryptographic signatures cannot be forged, this implies that the message was previously broadcast by σ. □

4.4 Validity

We now compute, given D and $\hat{D}$, the ϵ-**validity** of Contagion. To this end, we prove one preliminary lemma.

Lemma 28. *In an execution of Contagion, if pcb satisfies total validity and the sender has no more than $D - \hat{D}$ Byzantine processes in its delivery sample, then prb satisfies validity.*

Proof. Let m be the message broadcast by the correct sender σ. Since pcb satisfies total validity, every correct process eventually issues a `Ready`(m) message (i.e., a `Ready` message for m) (line 33).

By hypothesis, σ has no more than $D - \hat{D}$ Byzantine processes in its echo sample. Obviously, σ has at least $\hat{D}$ correct processes in its echo sample. Therefore, σ eventually receives at least $\hat{D}$ `Ready`(m) messages (line 37), and delivers m (line 56). □

Lemma 28 allows us to bound the ϵ-validity of Contagion, given D and $\hat{D}$.

Theorem 12. *Contagion satisfies ϵ_v-validity, with*

$$\begin{aligned} \epsilon_v &\leq \epsilon_v^{pcb} + \left(1 - \epsilon_v^{pcb}\right)\epsilon_o \\ \epsilon_o &= \sum_{\bar{F}=D-\hat{D}+1}^{D} Bin[D, f](\bar{F}) \end{aligned} \tag{4.1}$$

if the underlying abstraction of pcb satisfies ϵ_v^{pcb}-total validity.

Proof. We compute a bound on ϵ_v by assuming that, if the total validity of the underlying pcb instance is compromised, the validity of prb is compromised as well. Following from Lemma 28, the validity of prb can be compromised only if the total validity of pcb is compromised as well, or if σ has more than $D - \hat{D}$ Byzantine processes in its delivery sample.

Since procedure $sample$ independently picks D processes with replacement, each element of a correct process' echo sample has an independent probability f of being Byzantine, i.e., the number of Byzantine processes in a correct delivery sample is binomially distributed.

Therefore, σ has a probability ϵ_o of having more than $D - \hat{D}$ Byzantine processes in its delivery sample. □

4.5 Adversarial execution

In this section, we define the model underlying an adversarial execution of Contagion. Here, a Byzantine adversary is an agent that acts upon a system with the goal to compromise its consistency and / or totality. The main goal of this section is to formalize the information available to the adversary, and the set of actions that it can perform on the system throughout an adversarial execution.

Throughout the rest of this section, we bound the probability of compromising the consistency and totality of Contagion by assuming that, if the consistency of the pcb instance used in Contagion is compromised, then both the consistency and the totality of Contagion are compromised as well. In what follows, therefore, we assume that Sieve satisfies consistency.

4.5.1 Model

Let π be any correct process. We make the following assumptions about an adversarial execution of Contagion:

- As we established in Section 1.3, the adversary does not know which correct processes are in π's ready or delivery samples. The adversary knows, however, which Byzantine processes are in π's ready sample, and which Byzantine processes are in π's delivery sample.
- At any time, the adversary knows the set of messages for which π sent a `Ready` message.
- At any time, the adversary knows if π delivered a message. If π delivered a message, then the adversary knows which message did π deliver.
- The adversary can arbitrarily cause π to pcb.Deliver a given message m^*. Since we assume that the underlying pcb instance satisfies consistency, the adversary cannot cause two correct processes to pcb.Deliver two different messages.

Throughout an adversarial execution of Contagion, an adversary performs a sequence of minimal operations on the system. Each operation consists of either of the following:

- Selecting a correct process that did not pcb.Deliver m^* and causing it to pcb.Deliver m^*.
- Selecting a Byzantine process and causing it to issue a `Ready` message to a correct process.

As a result of each operation, zero or more processes send a Ready message and/or deliver a message. The adversary is successful if, at the end of the adversarial execution, either the consistency or the totality of the system is compromised.

4.6 Epidemic processes

In the next sections, we compute bounds for the ϵ-consistency and ϵ-totality of Contagion. In order to do so, in this section we study the *feedback mechanism* produced by Ready messages in an execution of Contagion.

As we discussed in Section 4.2, a correct process issues a Ready message for a message m after either pcb.Delivering m (line 33) or collecting at least $\hat{R}$ $\texttt{Ready}(m)$ messages from its ready sample (line 51). We formalize this observation in the following definition.

Definition 29 (Ready, E-ready, R-ready)**.** Let π be a correct process, let m be a message. Throughout an execution of Contagion, π is **E-ready** for m if π eventually pcb.Delivers m; π is **R-ready** for m if π eventually receives at least $\hat{R}$ $\texttt{Ready}(m)$ messages from its ready sample; π is **ready** for m if π is either E-ready or R-ready for m.

We note how a correct process can simultaneously be E-ready and R-ready for the same message.

It is easy to observe that the R-ready condition creates a feedback process: as a result of a correct process being R-ready for a message m, it issues a $\texttt{Ready}(m)$ message that might cause other correct processes to become R-ready for m as well.

Intuitively, this feedback process is designed to have two stable configurations:

- **Few processes are ready**: the fraction of correct processes that are E-ready for a message m is significantly smaller than $\hat{R}/R$. As a result, the probability of a correct process being R-ready for m becomes very small, and the set of processes that are ready for m is, with high probability, nearly identical to the set of processes that are E-ready for m.
- **All processes are ready**: the fraction of correct processes that are E-ready for a message m is not significantly smaller than $\hat{R}/R$. As a result, a correct process that is not E-ready for m has a significant probability of becoming R-ready for m. If this happens, the probability of a correct process becoming R-ready for m further increases, and eventually every correct proces is ready for m.

In this section, we show that the R-ready feedback mechanism is isomorphic to an epidemic process as we define it in Section 4.11. In summary, an epidemic process depends on one parameter (contagion threshold $\hat{R}$) to mimic the spread of a disease in a population:

- A population is represented on the nodes of a directed multigraph, allowing multi-edges and loops. Intuitively, an $a \rightarrow b$ edge represents the relation *a can infect b*.
- Each member of the population (or node) can be in either of two states: healthy or infected. An infected node stays infected: there is no cure for the infection.
- A set of nodes is initially infected. The epidemic process evolves in steps. At every step, all the nodes that have at least $\hat{R}$ infected predecessors become infected as well. The process is completed when either all nodes are infected, or no healthy node has at least $\hat{R}$ infected predecessors.

We refer the reader to Section 4.11 for a more formal discussion of epidemic processes. In this section, we prove the critical result that the R-ready feedback mechanism in Contagion is isomorphic to an epidemic process.

Definition 30 (Adversarial execution)**.** A **adversarial execution** (or just **execution**) is the sequence of events produced by an execution of Contagion on N processes, a fraction f of which are under the control of the adversary described in Section 4.5.1. For the sake of brevity, we omit a more formal definition.

Let x, x' be executions. We say that x is equivalent to x' ($x = x'$) if:

- The sequences of messages exchanged are identical in x and x'.
- The values produced by each correct, local source of randomness are identical in x and x'.

Definition 31 (Ready sample matrix)**.** A **ready sample matrix** is an element of the set

$$\mathscr{J} = \left(\Pi^R\right)^{\Pi_C}$$

Definition 32 (Ready sample matrix of an execution)**.** Let x be an execution, let j be a ready sample matrix. j **is** x**'s sample matrix** if, for every correct process π, π's ready sample in x is j_π.

Definition 33 (Random ready sample matrix)**.** A **random ready sample matrix** is a random variable representing the sample matrix of a random execution.

Lemma 29. *Random sample matrices are uniformly distributed. More formally, if $\bar{j}$ is a random sample matrix, then*

$$\mathscr{P}[\bar{j}] = \left(\frac{1}{N}\right)^{RC}$$

Proof. As we discussed in Section 1.3, the adversary has no control over the local source of randomness of each correct process. Each correct process independently selects with uniform probability R elements for its ready sample. □

Lemma 30. *Let j be a ready sample matrix. Let x, x' be executions of Contagion such that:*

- *No Byzantine process issues any* `Ready` *message in x or x'.*
- *The ready sample matrix of both x and x' is j.*

Let ρ_E, ρ'_E denote the set of correct processes that are E-ready for m in x, x' respectively. Let ρ, ρ' denote the set of correct processes that are ready for m in x, x' respectively.

We have

$$\left(\rho_E = \rho'_E\right) \implies \left(\rho = \rho'\right)$$

Proof. Let us assume $\rho_E = \rho'_E$. Let π be a correct process. As we established, π is ready for m if π is either E-ready or R-ready for m. Since $\rho_E = \rho'_E$, we immediately have that π is E-ready for m in x if and only if π is E-ready for m in x'.

By definition, π is R-ready for m in x (x') if it eventually receives at least $\hat{R}$ `Ready`(m) messages from its ready sample in x (x'). By hypothesis, no Byzantine process issues any `Ready` message in x (x'). Therefore, π is eventually R-ready for m in x (x') if π receives at least $\hat{R}$ `Ready`(m) messages from the correct processes in its ready sample in x (x').

As we discussed in Section 1.3, we assume that every message is eventually delivered in an unbounded but finite amount of time. Therefore, π is eventually R-ready for m in x (x') if at least $\hat{R}$ correct processes in π's sample eventually issue a `Ready`(m) message in x (x'), i.e., if at least $\hat{R}$ correct processes in π's sample are eventually ready for m in x (x').

Since the above condition does not depend on the network scheduling, a correct process π is eventually ready for m in x if and only if π is also eventually ready for m in x'. Therefore, $\rho = \rho'$. □

Lemma 31. *Let x be an execution of Contagion where no Byzantine process ever issues any* `Ready` *message. Let j be x's ready sample matrix. Let m be a message, let ρ_E denote the set of correct processes that are E-ready for m in x. Let ρ denote the set of correct processes that are eventually ready for m in x.*

Let $s_0 = ((v, e), w_0)$ be a contagion state (as defined in Definition 35), with

$$\begin{aligned} v &= \Pi_C \\ \left(\pi, \pi'\right) \in e &\iff \pi \in j_{\pi'} \\ w_0 &= \rho_E \end{aligned}$$

Let $s_\infty = ((v, e), w_\infty)$ be the contagion state resulting from the epidemic process with input s_0. We have

$$\rho = w_\infty$$

Proof. Following from Lemma 30, ρ does not depend on x's network scheduling. Without loss of generality, we can therefore make a synchrony assumption for x, and assume that every message delay is unitary.

Let ρ_t denote the set of correct processes that are ready for m in x at time t. We have

$$\rho_0 = w_0 = \rho_E$$

In x, a correct process that is not ready for m at time t becomes ready for m at time $t+1$ if at least $\hat{R}$ processes in its ready sample are ready for m at time t. Therefore

$$\pi \in \rho_{t+1} \iff \left(\pi \in \rho_t \lor \left|j_\pi \cap R_t\right| \geq \hat{R}\right)$$

As we discuss in Section 4.11, at step $t+1$, all the healthy nodes in an epidemic process that have at least $\hat{R}$ predecessors infected at time t become infected. Therefore

$$\pi \in w_{t+1} \iff \left(\pi \in w_t \lor \left|j_\pi \cap w_t\right| \geq \hat{R}\right)$$

Therefore, if $\rho_t = w_t$, then $\rho_{t+1} = w_{t+1}$, and, by induction, for all t, $\rho_t = w_t$. In Section 4.11, we prove that an epidemic process identically converges in a finite number of steps. Consequently, $\rho_\infty = w_\infty$, which proves the lemma. □

Lemma 32. *Let m be a message. Let x be an execution of Contagion where every Byzantine process sends a* `Ready`*(m) message to every correct process from which it received a* `ReadySubscribe` *message. Let j be x's ready sample matrix. Let ρ_E denote the set of correct processes that are E-ready for m in x. Let ρ denote the set of correct processes that are eventually ready for m in x.*

Let $s_0 = ((v, e), w_0)$ be a contagion state (as defined in Definition 35), with

$$\begin{aligned} v &= \Pi \\ (\pi, \pi') \in e &\iff (\pi' \in \Pi_C) \land (\pi \in j_{\pi'}) \\ w_0 &= \rho_E \cup (\Pi \setminus \Pi_C) \end{aligned}$$

Let $s_\infty = ((v, e), w_\infty)$ be the contagion state resulting from the epidemic process with input s_0. We have

$$\rho = w_\infty \setminus (\Pi \setminus \Pi_C)$$

Proof. It follows immediately from Lemma 31 and the observation that, in x, a Byzantine process sends the same `Ready` messages as a correct process that is E-ready for m. □

4.7 Threshold contagion (overview)

As we discussed in the previous section, in Section 4.11 we introduce epidemic processes, an abstract model of the feedback mechanism produced by `Ready` messages in an execution of Contagion. Given the multigraph on which it occurs, an epidemic process is deterministic. In Section 4.11, we also generalize epidemic processes to the probabilistic setting: we introduce and analyze Threshold Contagion, a game where a player infects in rounds arbitrary subsets of a population, causing a sequence of epidemic processes on a random, unknown multigraph.

Threshold Contagion depends on six parameters: node count N, sample size R, link probability l, round count K, infection batch S, and contagion threshold $\hat{R}$.

In summary, a game of Threshold Contagion is played as follows:

- A random multigraph with N nodes is generated. The number of predecessors of each node follows a $\text{Bin}[R, l]$ distribution. Each predecessor of a node is independently picked with uniform probability from the set of nodes.

  The topology of the network is not disclosed to the adversary.

- For K rounds:
  - The player infects an arbitrary set of S healthy nodes.
  - An epidemic process with contagion threshold $\hat{R}$ is ran on the resulting contagion state.

We refer the reader to Section 4.11 for a more formal discussion of Threshold Contagion. There we introduce the random variable

$$\gamma(N, R, l, K, S, \hat{R})$$

representing the number of nodes that are infected at the end of a game of Threshold Contagion. We then prove that, by arbitrary choosing which nodes to infect, the adversary has no way to bias γ. Finally, we analitically compute the probability distribution underlying γ.

In this section, we prove the critical result that a game of Threshold Contagion can be used to model two classes of adversarial executions of Contagion.

Lemma 33. *Let m^* be a message. Let x be an adversarial execution of Contagion where:*

- *No Byzantine process issues any* `Ready` *message.*
- *For K rounds:*
  - *The adversary selects, if possible, S correct process that are not ready for m^*, and causes them to pcb.Deliver m^*.*

– *The adversary waits until every resulting* `Ready` *message is delivered.*

Let ρ denote the number of correct processes in σ that, at the end of the adversarial execution, are ready for m. We have

$$\mathscr{P}[\bar{\rho}] = \mathscr{P}[\gamma(C, R, 1-f, K, S, \hat{R}) = \bar{\rho}]$$

Proof. We start by defining a function $\mathfrak{g} : \mathscr{J} \to \mathscr{G}$ (see Definition 39 for a definition of $\mathscr{G}$) by

$$\mathfrak{g}(j)_{i,k} = \begin{cases} \zeta^{-1}\left(j_{\zeta(i),k}\right) & \text{iff } j_{\zeta(i),k} \in \Pi_C \\ \bot & \text{otherwise} \end{cases}$$

We start by noting that, for every $\bar{g} \in \mathscr{G}$,

$$\mathscr{P}[\bar{g}] = \mathscr{P}[\mathfrak{g}^{-1}(g)]$$

Indeed, following from Lemmas 41 and 42:

$$\begin{aligned} \mathscr{P}[\bar{g}] &= \prod_{i,k} \mathscr{P}[\bar{g}_{i,k}] \\ \mathscr{P}[g_{i,k} = \bot] &= (1-l) = f \\ \mathscr{P}[g_{i,k} = (\bar{g}_{i,k} \in 1..C)] &= \frac{1}{C} \end{aligned}$$

and following from Definition 33 and Lemma 29 we have

$$\begin{aligned} \mathscr{P}[\bar{j}] &= \prod_{\pi,k} \mathscr{P}[\bar{j}_{\pi,k}] \\ \mathscr{P}[j_{\pi,k} \in \Pi \setminus \Pi_C] &= f \\ \mathscr{P}[j_{\pi,k} = (\bar{\pi}' \in \Pi_C)] &= \frac{1}{C} \end{aligned}$$

We now build from x a game of Threshold Contagion y, played on $\mathfrak{g}(j)$. At the beginning of each round, if the adversary causes a correct process π to pcb.Deliver m^*, then $\zeta(\pi)$ is infected.

We can prove that, if π is eventually ready for m^* in x, then $\zeta(\pi)$ is eventually infected in y. Indeed, following from Lemma 31, if π is ready for m^* at the end of a round in x, then $\zeta(\pi)$ is infected at the end the same round in y.

Therefore, the following hold true:

- The probability of $\bar{j}$ is identical to the probability of $\mathfrak{g}(\bar{j})$.
- The number of correct processes that are eventually ready for m^* in x is identical to the

number of nodes that are eventually infected in y.

□

Lemma 34. *Let m be a message. Let x be an adversarial execution of Contagion where:*

- *No correct process pcb.Delivers m.*
- *Every Byzantine process sends a* `Ready` *(m) message to every correct process from which it received a* `ReadySubscribe` *message.*

Let ρ denote the number of correct processes in σ that, at the end of the adversarial execution, are ready for m. We have

$$\mathscr{P}\left[\bar{\rho}\right] = \mathscr{P}\left[\gamma(N, R, 0, 1, N - C, \hat{R}) = \bar{\rho} + (N - C)\right]$$

Proof. The proof is similar to the proof of Lemma 33, using Lemma 32 instead of Lemma 31.

□

4.8 Preliminary lemmas

In order to compute an upper bound for the probability of the consistency of Contagion being compromised, we will make use of some preliminary lemmas. The statements of those lemmas are independent from the context of Contagion. For the sake of readability, we therefore gather them in this section, and use them throughout the rest of this section.

Lemma 35. *Let $N, K \in \mathbb{N}$ such that $K \in 0..N$. Let X be a random variable defined by*

$$\mathscr{P}\left[\bar{X}\right] = Bin\left[N, p\right]\left(\bar{X}\right)$$

We have that

$$\mathscr{P}[X \geq K]$$

is an increasing function of p.

Proof. We expand

$$\mathscr{P}[X \geq K] = \sum_{\bar{X}=K}^{N} \text{Bin}\left[N, p\right]\left(\bar{X}\right)$$

and take the derivative

$$\begin{aligned}
&\frac{\partial}{\partial p}\sum_{\bar{X}=K}^{N}\binom{N}{\bar{X}}p^{\bar{X}}(1-p)^{N-\bar{X}}\\
=&\sum_{\bar{X}=K}^{N}\binom{N}{\bar{X}}\left(p^{\bar{X}}(1-p)^{N-\bar{X}}\right)\left(\frac{\bar{X}}{p}-\frac{N-\bar{X}}{1-p}\right)\\
=&\underbrace{\frac{1}{p(1-p)}}_{\geq 1}\sum_{\bar{X}=K}^{N}\mathrm{Bin}[N,p](\bar{X})(\bar{X}-pN)\\
\geq&\sum_{\bar{X}=K}^{N}\mathrm{Bin}[N,p](\bar{X})(\bar{X}-pN)
\end{aligned}$$

We now prove that, for every $K\in[0,N]$,

$$\sum_{\bar{X}=K}^{N}\mathrm{Bin}[N,p](\bar{X})(\bar{X}-pN)\geq 0 \tag{4.2}$$

We start by noting that Equation (4.2) holds true for every $K>pN$. Indeed, if $K>pN$, then every term of the sum in Equation (4.2) is positive.

We prove that Equation (4.2) holds true for every $K<pN$ by induction. For $K=0$ we have

$$\begin{aligned}
&\sum_{\bar{X}=0}^{N}\mathrm{Bin}[N,p](\bar{X})(\bar{X}-pN)\\
=&\underbrace{\sum_{\bar{X}=0}^{N}\bar{X}\mathrm{Bin}[N,p](\bar{X})}_{=pN}-pN\underbrace{\sum_{\bar{X}=0}^{N}\mathrm{Bin}[N,p](\bar{X})}_{=1}\\
=&\ 0
\end{aligned}$$

Let us assume that Equation (4.2) holds true for some $K<pN$. We have

$$\begin{aligned}
&\sum_{\bar{X}=K+1}^{N}\mathrm{Bin}[N,p](\bar{X})(\bar{X}-pN)\\
=&\underbrace{\sum_{\bar{X}=K}^{N}\mathrm{Bin}[N,p](\bar{X})(\bar{X}-pN)}_{\geq 0\text{ by IH}}-\mathrm{Bin}[N,p](K)(K-pN)\\
\geq&\ 0
\end{aligned}$$

which proves that Equation (4.2) holds true for $K+1$ as well. By induction, Equation (4.2) holds true for every $K<pN$. This proves that the derivative is positive for all $p\in[0,1]$ which proves the lemma. □

4.9 Consistency

In this section, we compute a bound on the ϵ-consistency of Contagion. As we discussed in Section 4.5.1, here we bound the probability of compromising the consistency of Contagion by assuming that, if the consistency of the pcb instance used in Contagion is compromised, the consistency of Contagion is compromised as well.

Let m^* denote the only message that any correct process can pcb.Deliver. We start by noting that, simply by having every Byzantine process behave like a correct process, an adversary can cause any correct process to deliver m^*: indeed, with $f = 0$, Contagion satisfies validity deterministically [1].

As we discussed in Section 4.2, a correct process can issue a Ready message for an arbitrary number of messages. In other words, causing a correct process to become E-ready for m^* does not affect its behavior with respect to a message $m \neq m^*$.

Therefore, if an adversary can cause at least one correct process π to eventually receive at least $\hat{R}$ Ready messages for a message $m \neq m^*$, it can also compromise the consistency of Contagion.

Indeed, as we discussed in Section 1.3, the adversary has arbitrary control over the network scheduling. Even if π would eventually receive enough Ready(m^*) messages to deliver m^*, the adversary can slow those messages down, and cause π to first receive enough Ready(m) messages to deliver m. Every other correct process will eventually deliver m^*, thus compromising the consistency of the system.

We formalize the above intuition in the following lemma.

Lemma 36. *Let m^* denote the only message that any correct process can pcb.Deliver. An optimal adversary causes every Byzantine process to send a* Ready*(m) message, for some $m \neq m^*$, to every correct process from which it received a* ReadySubscribe *message.*

Proof. Let B denote the number of Byzantine processes that eventually issue a Ready(m) message. Let π be a correct process, let Q denote the number of Ready(m) messages that π eventually collects. Since π picks each element of its delivery sample independently, Q is binomially distributed:

$$\mathscr{P}[\bar{Q}] = \mathrm{Bin}\left[D, \frac{B}{N}\right](\bar{Q})$$

Following from Lemma 35,

$$\mathscr{P}[Q \geq \hat{D}] = \sum_{\bar{Q}=\hat{D}}^{D} \mathscr{P}[\bar{Q}]$$

[1] Here we are slightly abusing the result of Theorem 12, as it only guarantees that a correct sender will eventually deliver its message. The result, however, independently holds for any other correct process as well.

is an increasing function of B, and maximized by $B = (N - C)$. Therefore, the probability of π eventually receiving enough `Ready`(m) messages to deliver m is maximized if every Byzantine process issues a `Ready`(m) message.

As we previously established, the adversary can cause every correct process to also receive at least $\hat{D}$ `Ready`(m^*) messages. Since the adversary has control over network scheduling, it can cause π to deliver m, and every other process to deliver m^*, thus compromising the consistency of the system. □

Lemma 37. *Let m^* denote the only message that any correct process can pcb.Deliver, let $m \neq m^*$. If, throughout an optimal adversarial execution, no correct process eventually collects enough* `Ready`*(m) messages to deliver m, then no correct process eventually collects enough* `Ready`*(m) messages to deliver any message $m' \neq m$.*

Proof. Following from Lemma 36, the optimal adversary causes every Byzantine process to issue a `Ready`(m) message. In Lemma 32, we use the fact that this strategy makes the Byzantine processes behave identically to correct processes that are E-ready for m to show that the set of correct processes that are eventually ready for m only depends on the ready sample matrix of the execution.

Since a correct process does not change its ready or delivery samples throughout an execution, the set of processes that will eventually be ready for m' is at most the same as the set of processes that will eventually be ready for m. In turn, this means that if no correct process eventually delivers m, no correct process eventually delivers m' either. □

We can now use Lemma 36 to compute a bound on the ϵ-consistency of Contagion.

We introduce the random variable γ^+ by

$$\mathscr{P}\left[\bar{\gamma}^+\right] = \mathscr{P}\left[\gamma(N, R, 0, 1, N - C, \hat{R}) = \bar{\gamma}^+\right]$$

Following from Lemmas 34 and 36, γ^+ represents the number of processes (Byzantine or correct) that eventually issue a `Ready` message for a message $m \neq m^*$, when an optimal adversary is trying to compromise the consistency of the system.

We can finally compute a bound for the ϵ-consistency of Contagion. We define

$$\begin{aligned} \mu &= \sum_{\bar{\gamma}^+ = N-C}^{N} \left(1 - \left(1 - \tilde{\mu}(\bar{\gamma}^+)\right)^C\right) \mathscr{P}\left[\bar{\gamma}^+\right] \\ \tilde{\mu}(\bar{\gamma}^+) &= \sum_{\bar{D} = \hat{D}}^{D} \mathrm{Bin}\left[D, \frac{\bar{\gamma}^+}{N}\right](\bar{D}) \end{aligned}$$

Theorem 13. *Contagion satisfies ϵ_c-consistency, with*

$$\epsilon_c \leq \epsilon_c^{pcb} + \left(1 - \epsilon_c^{pcb}\right)\mu$$

if the underlying abstraction of pcb satisfies ϵ_c^{pcb}-consistency.

Proof. We start by noting that $\tilde{\mu}(\bar{\gamma}^+)$ represents the probability that a specific correct process will eventually collect enough $\texttt{Ready}(m)$ messages to deliver m, given the number $\bar{\gamma}^+$ of processes that eventually issue a $\texttt{Ready}(m)$ message.

Indeed, since every correct process picks its delivery sample independently, each of the D elements of a correct process' delivery sample has a probability $\bar{\gamma}^+/N$ of issuing a $\texttt{Ready}(m)$ message.

We then note that μ represents the probability of any correct process eventually collecting enough $\texttt{Ready}(m)$ messages to deliver m. μ is obtained by applying the law of total probability to $\mu(\bar{\gamma}^+)$.

Finally, ϵ_c is obtained by the assumption that, if the consistency of the underlying pcb instance is compromised, the totality of Contagion is compromised as well. □

4.10 Totality

In this section, we compute a bound on the ϵ-totality of Contagion. As we discussed in Section 4.5.1, here we bound the probability of compromising the totality of Contagion by assuming that, if the consistency of the pcb instance used in Contagion is compromised, the consistency of Contagion is compromised as well.

4.10.1 Minimal operations

Let m^* be the only message that any correct process can pcb.Deliver. As we discussed in Section 4.5.1, throughout an execution of Contagion, an adversary performs a sequence of minimal operations on the system, i.e., it either causes a correct process to pcb.Deliver m^*, or it causes a Byzantine process to send an arbitrary $\texttt{Ready}(m)$ message to a correct process.

We further relax the bound by assuming that, if the adversary can cause any message $m \neq m^*$ to be delivered by at least one correct process, the totality of Contagion is compromised as well.

Under the assumption that no correct process can eventually collect enough $\texttt{Ready}(m)$ messages to deliver any message m different from m^*, causing a Byzantine process to send a $\texttt{Ready}(m)$ message has no effect on the totality of the system.

This reduces the set of adversarial operations that have a non-null effect on the totality of the system to:

- Causing an arbitrary correct process to pcb.Deliver m^*.
- Causing a Byzantine adversary to send a `Ready`(m^*) message to a correct process.

We now prove a lemma to further reduce the set of minimal operations of an optimal adversary.

Lemma 38. *Let m^* be the only message that any correct process can potentially pcb.Deliver. Let π be a correct process, let ξ be a Byzantine process in π's ready sample. An optimal adversary never causes ξ to send a* `Ready`*(m^*) message to π.*

Proof. As a result of receiving a `Ready`(m^*) message from ξ, π can either:

- Have collected less than $\hat{R}$ `Ready`(m^*) messages from its ready sample. The operation has no effect.
- Have collected exactly $\hat{R}$ `Ready`(m^*) messages from is ready sample. Then π becomes ready for m^*. However, the same outcome could have been achieved deterministically by causing π to pcb.Deliver m^*.

Since every outcome of ξ's `Ready`(m^*) message to π can be deterministically emulated by causing π to pcb.Deliver (or not pcb.Deliver) m^*, the operation is useless to an optimal adversary. □

4.10.2 Delivery probability

Let γ^- denote the random variable counting the number of correct processes that are eventually ready for m^*. In this section, we study the probability of totality being compromised, given the value of γ^-.

By definition, totality is compromised if at least one correct process delivers m^* and one correct process does not deliver m^*.

Let π be a correct process. We introduce the following events:

- A_π: process π delivers m^*.
- A: all correct processes deliver m^*.
- $\tilde{A}$: no correct process delivers m^*.
- T: the totality of the system is compromised.

Given $\bar{\gamma}^-$, the probability of A_π is bound by

$$\alpha_\pi^-(\bar{\gamma}^-) \leq \mathscr{P}\left[A_\pi \mid \bar{\gamma}^-\right] \leq \alpha_\pi^+(\bar{\gamma}^-)$$

with

$$\begin{aligned}
\alpha_\pi^-(\bar{\gamma}^-) &= \sum_{\bar{D}=\hat{D}}^{D} \mathrm{Bin}\left[D, \frac{\bar{\gamma}^-}{N}\right](\bar{D}) \\
\alpha_\pi^+(\bar{\gamma}^-) &= \sum_{\bar{D}=\hat{D}}^{D} \mathrm{Bin}\left[D, \frac{\bar{\gamma}^- + (N-C)}{N}\right](\bar{D})
\end{aligned}$$

The lower bound is attained when none of the Byzantine processes issue a `Ready`(m^*) message, and the upper bound is attained when all Byzantine processes issue a `Ready`(m^*) message.

Noting that each correct process independently picks its delivery sample, we can compute, given $\bar{\gamma}^-$, a lower bound for the probability of A:

$$\mathscr{P}\left[A \mid \bar{\gamma}^-\right] \geq \left(\alpha^-(\bar{\gamma}^-)\right)^C$$

and a lower bound for the probability of $\tilde{A}$:

$$\mathscr{P}\left[\tilde{A} \mid \bar{\gamma}^-\right] \geq \left(1 - \alpha^+(\bar{\gamma}^-)\right)^C$$

The above allow us to compute, given $\bar{\gamma}^-$, an upper bound for the probability of T:

$$\begin{aligned}
\mathscr{P}\left[T \mid \bar{\gamma}^-\right] &= \mathscr{P}\left[\not{A}, \not{\tilde{A}} \mid \bar{\gamma}^-\right] \\
&\leq 1 - \mathscr{P}\left[A \mid \bar{\gamma}^-\right] - \mathscr{P}\left[\tilde{A} \mid \bar{\gamma}^-\right] \\
&\leq \alpha(\bar{\gamma}^-)
\end{aligned}$$

with

$$\alpha(\bar{\gamma}^-) = 1 - \left(\alpha^-(\bar{\gamma}^-)\right)^C - \left(1 - \alpha^+(\bar{\gamma}^-)\right)^C$$

4.10.3 C-step Threshold Contagion

Due to Lemma 38, the minimal set of operations for an optimal adversary reduces to

- Causing an arbitrary correct process to pcb.Deliver m^*.
- Causing a Byzantine process ξ in the delivery sample of a correct process π to send a `Ready`(m^*) message to π.

It is immediate to see that the latter operation has no effect over which correct processes eventually become `Ready` for m^*. In the previous section, we computed an upper bound

on the probability of compromising the totality of the system, given the number of correct processes that are eventually ready for m^*.

In this section, we prove a final constraint on the optimal adversarial strategy, and finally compute a bound on the ϵ-totality of Contagion.

Lemma 39. *Let m^* denote the only message that any correct process can pcb.Deliver. An optimal adversary executes in C rounds. At every round, the adversary causes one correct process to pcb.Deliver m^*, then waits until all the resulting* `Ready` *messages are delivered.*

Proof. Due to Lemma 31, the outcome of the execution is not affected by network scheduling: causing one correct process at a time to pcb.Deliver m^* has the same effect, e.g., as causing any set of correct processes to simultaneously pcb.Deliver m^*. □

Following from Lemma 39, we can intuitively see an adversarial execution whose goal is to compromise the totality of Contagion as a game similar to *blackjack*. The game unfolds in C rounds. At every round, the adversary causes one more correct process to pcb.Deliver m^*. With high probability, this will have two possible negative outcomes for the player:

- Nothing happens: no correct process is able to deliver m^*, even if the Byzantine processes in its delivery sample issue a `Ready`(m^*) message. The only possible move is to play again.
- The execution is *busted*: a feedback loop is generated that eventually causes, with high probability, every correct process to deliver m^*, even if no Byzantine process issues any `Ready`(m^*) message. The adversary fails in compromising the totality of the system.

If the adversary is lucky enough, however, one of the rounds will result in a configuration where no feedback loop occurred, but at least one correct process can deliver m^*. In that case, the adversary causes that process to deliver m^*, and stops: totality is compromised.

Following from Lemma 33, the probability distribution underlying the number of correct processes that are ready for m^* at the end of the n-th step is

$$\mathscr{P}[\bar{\gamma}_n^-] = \mathscr{P}[\gamma(C, R, 1-f, C, 1, \hat{R})]$$

We can finally compute a bound on the ϵ-totality of Contagion.

Theorem 14. *Contagion satisfies ϵ_t-totality, with*

$$\epsilon_t \le \epsilon_c^{pcb} + \mu + \epsilon_b$$
$$\epsilon_b = \sum_{n=0}^{C} \sum_{\bar{\gamma}_n^-=0}^{C} \mathscr{P}[\gamma_n^-]\alpha(\bar{\gamma}_n^-)$$

if the underlying abstraction of pcb satisfies ϵ_c^{pcb}*-consistency.*

Proof. Let T_n denote the event of totality being compromised at the end of round T_n.

Under the assumption that the consistency of pcb is satisfied, and no message other than m^* can be delivered by any correct process, the probability of T_n with $n > 1$ is

$$\mathscr{P}[T_n] = \sum_{\bar{\gamma}_n^-=0}^{C} \mathscr{P}\left[T_n \mid \bar{\gamma}_n^-\right] \mathscr{P}\left[\bar{\gamma}_n^- \mid \not{T}_{n-1}\right]$$

Indeed, the adversary will proceed to round n only if round $n-1$ was unsuccessful in compromising the totality of the system. We can use the law of total probability to get

$$\begin{aligned} \mathscr{P}[T_n] &\leq \sum_{\bar{\gamma}_n^-=0}^{C} \mathscr{P}\left[T_n \mid \bar{\gamma}_n^-\right]\left(\mathscr{P}\left[\bar{\gamma}_n^- \mid \not{T}_{n-1}\right] + \mathscr{P}\left[\bar{\gamma}_n^- \mid T_{n-1}\right]\right) \\ &= \sum_{\bar{\gamma}_n^-=0}^{C} \mathscr{P}\left[T_n \mid \bar{\gamma}_n^-\right] \mathscr{P}\left[\bar{\gamma}_n^-\right] \end{aligned}$$

We can use Boole's inequality to get

$$\mathscr{P}[T] \leq \sum_{n=0}^{C} \mathscr{P}[T_n]$$

and since

$$\mathscr{P}\left[T_n \mid \bar{\gamma}_n^-\right] \leq \alpha(\gamma_n^-)$$

we have that ϵ_b bounds the probability of compromising totality, if the consistency of pcb is satisfied, and no message other than m^* can be delivered by any correct process.

The value provided for ϵ_t follows from applying again Boole's inequality to include ϵ_c^{pcb} and μ (which, in Section 4.9, we proved to bound the probability of any correct process delivering a message other than m^*). □

4.11 Threshold Contagion

In this section we discuss *epidemic processes*, mimicking the spread of a disease in a population, and the Threshold Contagion game, which gives a player the possibility to actively infect parts of a population.

As we discuss in Section 4.2, in `Contagion`, when a correct process receives enough `Ready` messages from its *ready sample* for the same message m, it issues itself a `Ready` message for m. This produces a feedback mechanism that, in Section 4.7, we showed to be isomorphic to an epidemic process as we define it below.

Threshold Contagion is a game where a player iteratively applies the epidemic process to chosen inputs. We use Threshold Contagion for modeling and analyzing our Contagion algorithm.

4.11.1 Epidemic processes

An epidemic process models the spreading of a disease in a population.

Preliminary definitions

Definition 34 (Directed multigraph)**.** A **directed multigraph** is a pair $g = (v, e)$, where v is a set and $e : v^2 \to \mathbb{N}$ is a multiset whose elements are pairs of elements of v. We call the elements of v the **vertices** (or **nodes**) of g. We call the elements of e the **edges** of g.

Following from Definition 34, a directed multigraph allows self-loops (let $a \in v$, (a, a) can be an element of e) and multiple edges (let $a, b \in v$, the multiplicity of (a, b) in e can be greater than one).

Contagion state

The spreading of a disease is represented by a **contagion state**.

Definition 35 (Contagion state)**.** A **contagion state** is a pair $s = (g, w)$, where $g = (v, e)$ is a multigraph and $w \in \mathbb{P}(v)$. We call the elements of w the **infected nodes** of s.

Let $s = ((v, e), w)$ be a contagion state. In an epidemic process:

- Each node in v corresponds to one individual member of the population.
- A node is always in one of two possible states: **healthy** or **infected**. We do not consider any cure—once a node becomes infected, it stays infected forever. The set w represents the nodes that are infected.
- Edges model interactions between the members of the population. The multiset of edges e represents the "can infect" relation. Note that this relation is not symmetric. A directed edge $(a \to b)$ between nodes a and b means that a can infect b, but not that b can infect a.

Definition 36 (Predecessors)**.** Let $g = (v, e)$ be a multigraph, let $a \in v$. Then the **predecessors** of a in g form the multiset $\mathfrak{p}[a] : v \to \mathbb{N}$ defined by

$$\mathfrak{p}[a](x \in v) = e(x, a)$$

Following from Definition 36, the predecessors of a node a in a multigraph g form the multiset of nodes that have an edge to a. If a node has multiple edges to a, then it has a multiplicity greater than one in $\mathfrak{p}[a]$.

Contagion rule

In an epidemic process, the infection of healthy nodes follows a single rule.

- **Contagion rule**: A healthy node becomes infected if the number of its infected predecessors reaches a critical threshold.

The input to an epidemic process is a contagion state s. The epidemic process repeatedly applies the contagion rule to s until either all nodes are infected or no healthy node has enough infected predecessors to become infected itself. The epidemic process outputs the resulting contagion state.

4.11.2 Threshold Contagion

Threshold Contagion is a game played on the nodes of a random directed multigraph g. Threshold Contagion consists of one or more **rounds**. Each round inputs a contagion state s and outputs a contagion state s'. The input to the first round is the contagion state $(g, \emptyset)$, i.e., the contagion state with no infected nodes whose multigraph is g. The input to any other round is the output of the previous round.

A round with input s is played as follows:

- The **player** infects a fixed-size subset of the healthy nodes of the contagion state s. This results in a contagion state s'.
- The contagion state s' is provided as input to an epidemic process. The output s'' of the epidemic process is returned.

4.11.3 Rules

In this section, we formally define the rules of Threshold Contagion and introduce its **parameters**.

Threshold Contagion is played on the nodes of a random, directed multigraph $g = (v, e)$. The in-degree of each node n in v is independently binomially distributed; each predecessor of n is uniformly picked with replacement from v.

Parameters

A game of Threshold Contagion depends on the following numerical parameters:

- **Node count** ($N \in \mathbb{N}$): Represents the number of nodes in the multigraph ($N = |v|$).
- **Sample size** ($R \in \mathbb{N}$): Represents the maximum in-degree of a node in the multigraph.
- **Link probability** ($l \in [0,1]$): Represents the probability of a predecessor link being successfully established. The in-degree of a node follows the distribution $\text{Bin}[R, l]$.
- **Round count** ($K \in \mathbb{N}_{>0}$): Represents the number of rounds in the game.
- **Infection batch** ($(S < N) \in \mathbb{N}_{>0}$): Represents the number of healthy nodes the player infects at the beginning of each round.
- **Contagion threshold** ($(\hat{R} \leq R) \in \mathbb{N}$): Represents the number of infected predecessors that will cause an healthy node to become infected (see Contagion rule).

Game

A game of Threshold Contagion is played as follows:

- A random, directed multigraph $g = (v, e)$ with N nodes is built. For every node n in v:
  - R times:
    - A Bernoulli random variable $\bar{B} \leftarrow \text{Bern}[l]$ is sampled.
    - If $\bar{B} = 1$, then a random node m is selected with uniform probability from v, and the edge $m \to n$ is added to e (i.e., m is added to the predecessors of n).
- Let $s = (g, w = \emptyset)$ be a contagion state. For K **rounds**:
  - If at least S nodes in v are healthy (i.e., they are not in the set of infected nodes w), the player selects S distinct nodes and infects them. The player **cannot** inform this choice with knowledge of the topology of g.
  - An epidemic process is run on s: until either every node in v is infected (i.e., $v = w$), or no healthy node in v has at least $\hat{R}$ infected predecessors, the following **contagion step** is iterated:
    - Every node in v with at least $\hat{R}$ infected predecessors is infected, i.e., it is added to w.

Figure 4.1 shows an example game of Threshold Contagion with small parameters.

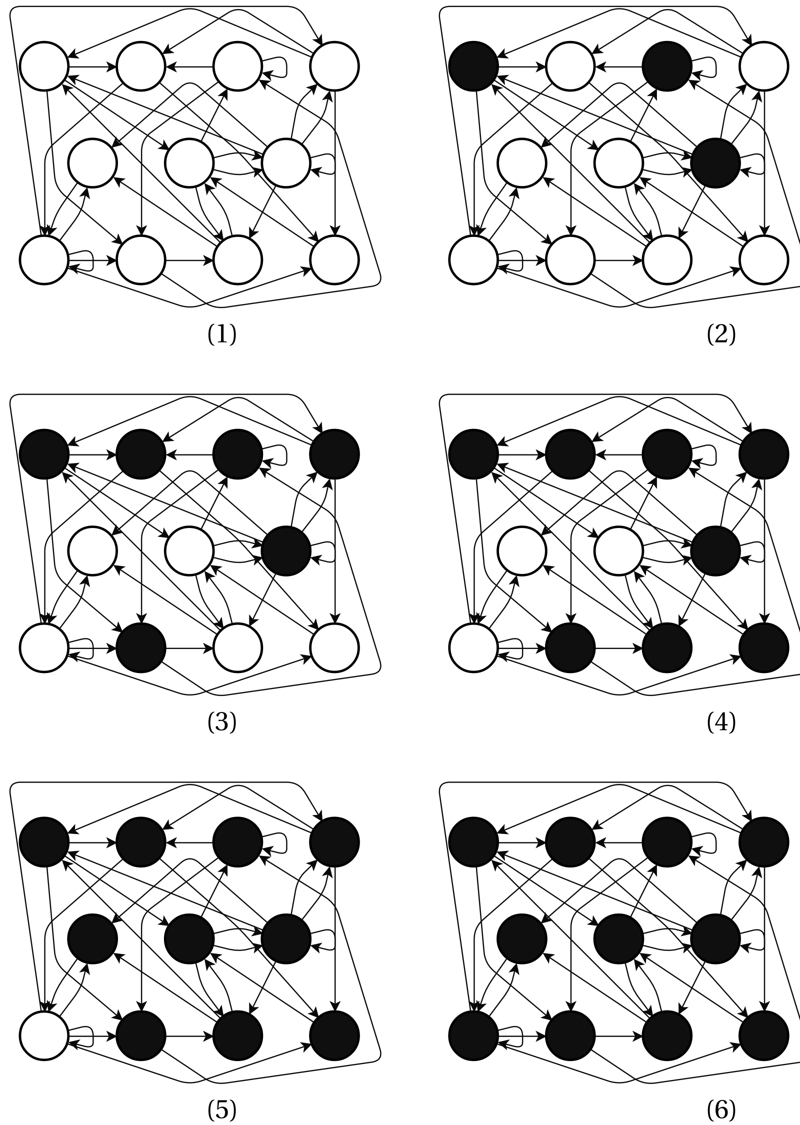


Figure 4.1: An example game of Threshold Contagion. Here $N = 11$, $l = 1$, $R = 3$, $\hat{R} = 2$, $K = 1$ and $S = 3$. Notice how nodes can be linked to themselves, form loops, or be linked more than once. An initial set of S nodes (1) is infected by the player (2). The game then unfolds in contagion steps (3 to 6): whenever a node has at least $\hat{R}$ infected predecessors, it becomes infected. This example shows how easily a game of Threshold Contagion can converge to a fully-infected configuration.

4.11.4 Random variables

We introduce the following random variables, which we discuss in more formal detail in the next sections:

- **Infection size** N_i^r: represents the number of infected nodes at round r and step i.
- **Frontier size** U_i^r: represents the number of nodes that are infected at round r and step i, but are not infected at round r and step $i-1$.
- **Infection status** $W_i^r[j]$: represents whether or not the j-th node is infected at round r and step i. We use $W_i^r[j]$ to signify that the node is infected, and $\not{W_i^r[j]}$ to signify that the node is not infected.
- **Infected predecessors count** $V_i^r[j]$: represents the number of predecessors of the j-th node that are infected at round r and step i.

Remark: for the sake of readability, the round number and/or the node index (for W and V) will be omitted whenever it can be unequivocally inferred from the context.

4.11.5 Goal

The goal of this section is to compute the probability distribution underlying the number of infected nodes at the end of a game of Threshold Contagion.

Lemma 40. *For any r, the random variables N_i^r, U_i^r, $W_i^r[j]$, and V_i^r converge in a finite number of steps.*

Proof. We note the following:

- N_i^r is a non-decreasing function of i, and $N_i^r \leq N$.
- $U_{i>0}^r = N_i^r - N_{i-1}^r$.
- For any j, $W_i^r[j] \implies W_{i+1}^r[j]$.
- V_i^r is a non-decreasing function of i and $V_i^r \leq R$.

From the above follows that all random variables converge for $i \to \infty$.

The codomains of N, U, W and V are all finite. Therefore, all random variables converge in a finite number of steps. □

Corollary 2. *All rounds terminate in a finite number of contagion steps.*

Notation 10 (End of round)**.** We use N^r_∞, U^r_∞, $W^r_\infty[j]$, V^r_∞ to denote the values of N, U, W, V at the end of round r.

The goal of this section is to compute the probability distribution underlying the random variable

$$\gamma(N,R,l,K,S,\hat{R}) = N^K_\infty \tag{4.3}$$

i.e., the probability of a game of Threshold Contagion resulting in $\bar{N}^K_\infty$ nodes ultimately being infected. Lemma 40 proves that Γ is a well defined variable (i.e., the limit exists) and, since K is finite, can be computed in a finite total number of steps.

4.11.6 Sample space

In this section, we define a sample space for Threshold Contagion, i.e., the set of all possible outcomes of a Threshold Contagion game. As we described in Section 4.11.2, the outcome of a game of Threshold Contagion is completely determined by two factors:

1. The topology of the random multigraph g on which Threshold Contagion is played. The probability distribution underlying g is known, and we compute it in this section.
2. The player's infection strategy, i.e., the nodes the player chooses to infect at the beginning of each round. The probability distribution underlying the player's choices is **unknown** and arbitrary. In this section, we only formalize their sample space.

Thus, an element of the sample space is a pair consisting of a multigraph (1.) and an infection strategy (2.).

Multigraph

As discussed in Section 4.11.3, a game of Threshold Contagion is played on the nodes of a multigraph $g = (v, e)$ allowing multi-edges and loops. Every node in v has at most R predecessors. Therefore, g can be represented by a **predecessor matrix** as we define it below.

We start by explicitly labeling the elements of v.

Notation 11 (Vertices)**.** Let $g = (v, e)$ be a multigraph, with $|v| = N$. Without loss of generality, we label the elements of v using natural numbers:

$$v = 1..N$$

Since every node in g has at most R predecessors, for every $j \in v$ we can represent the elements of $\mathfrak{p}(j)$ as the components of a *predecessor vector*.

Definition 37 (Predecessor vector)**.** A **predecessor vector** is an element of the set

$$\mathscr{R} = (\{\perp\} \cup v)^R$$

In a multigraph $g = (v, e)$, whose in-degree is bound by R, we use a predecessor vector to represent the predecessors of a node. Let $r \in \mathscr{R}$ be the predecessor vector of a node $j \in v$. If $r_k = \perp$, we say that the k-th predecessor of j is **missing**.

As discussed in Section 4.11.3, the predecessors of each node in v are generated by independently sampling R times a value $\bar{B}$ from a Bernoulli variable; whenever $\bar{B} = 1$, an additional predecessor is uniformly picked with replacement from the elements of v. We call a vector of predecessors selected this way a *random predecessor vector*, as formally defined in Definition 38.

Definition 38 (Random predecessor vector)**.** A **random predecessor vector** is a predecessor vector generated by the procedure described in Section 4.11.3.

Let r be a random predecessor vector. For every $k \in \{1, \ldots, R\}$, $\bar{B} \leftarrow \text{Bern}[l]$ is independently sampled; if $\bar{B} = 0$, r_k is set to $\perp$, otherwise r_k is set to an element of v, picked independently with uniform probability.

Lemma 41. *Let r be a random predecessor vector. Then*

$$\begin{aligned} \mathscr{P}[\bar{r}] &= \prod_{k=1}^{R} \mathscr{P}[\bar{r}_k] \\ \mathscr{P}[r_k = \perp] &= (1 - l) \\ \mathscr{P}[r_k = (\bar{r}_k \in v)] &= \frac{l}{N} \end{aligned}$$

Proof. Following from Definition 38, each component of r is independently sampled. Each component has a probability $(1 - l)$ of being missing. Each non-missing component of r has an equal probability of being equal to any element of v. □

As we discussed in Section 4.11.3, the multigraph $g = (v, e)$ is constructed by independently generating the predecessors for each node in v. Therefore, the topology of g is completely determined by N predecessor vectors, that can be organized in a *predecessor matrix*.

Definition 39 (Predecessor matrix)**.** A **predecessor matrix** is an element of the set

$$\mathscr{G} = \mathscr{R}^N$$

Notation 12 (Predecessor matrix)**.** Since a predecessor matrix uniquely identifies a multigraph, we interchangebly use g to denote a predecessor matrix and its corresponding multigraph. Let g be a predecessor matrix defining a multigraph (v, e), then g_j is the predecessor vector of node $j \in v$.

Definition 40 (Random predecessor matrix)**.** A **random predecessor matrix** is a predecessor matrix representing the outcome of the multigraph generation process described in Section 4.11.3. More formally, a random predecessor matrix consists of N independent random predecessor vectors.

Lemma 42. *Let g be a random predecessor matrix. Then*

$$\mathscr{P}[\bar{g}] = \prod_{j=1}^{N} \mathscr{P}[\bar{g}_j] \tag{4.4}$$

Proof. It follows immediately from Definition 40. □

Sub-threshold predecessor set

As discussed in Section 4.11.3, an epidemic process consists of a sequence of contagion steps. Let $s = ((v, e), w)$ be a contagion state. In a contagion step, a healthy node $j \in v$ ($j \in w$) becomes infected if at least $\hat{R}$ of its predecessors are infected, i.e., if

$$\left|\mathfrak{p}(j) \cap w\right| \geq \hat{R}$$

Given the set w, the set of predecessor vectors that do not satisfy the condition above is uniquely defined. We define *sub-threshold predecessor sets* to capture this notion.

Definition 41 (Sub-threshold predecessor set)**.** Let g be a predecessor matrix defining a multigraph (v, e). Let $X \subseteq v$. The **sub-threshold predecessor set** of X is the set

$$\tilde{\mathscr{R}}^X = \left\{r \in \mathscr{R} \mid |\{k \in 1..R \mid r_k \in X\}| < \hat{R}\right\}$$

$\tilde{\mathscr{R}}^X$ contains all the predecessor vectors in $\mathscr{R}$ that have less than $\hat{R}$ components in X.

Figure 4.2 shows an example multigraph where the predecessors of three nodes are displayed, two of which are in the sub-threshold predecessor set of a given set X.

Player's strategy

As discussed in Section 4.11.3, at the beginning of each round of Threshold Contagion the player selects, if possible, S distinct healthy nodes and infects them. These are the only K choices the player makes throughout Threshold Contagion. Moreover, the player has no knowledge of the topology of the multigraph g on which Threshold Contagion is played.

The player's choices can be expressed in an *infection strategy*, as we formally define it in this section. Together with the topology of the multigraph on which the game is played, an infection strategy uniquely determines the outcome of an instance of Threshold Contagion.

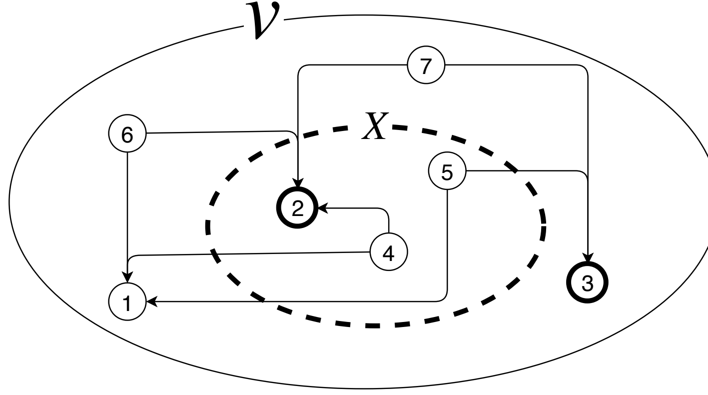


Figure 4.2: An example multigraph $g = (v, e)$ with 7 nodes. A subset $X \subseteq v$ is highlighted. Numbered dots represent the elements of v, and the edges to nodes 1, 2 and 3 are displayed. With $R = 3$ and $\hat{R} = 2$, we have $g_1 \notin \tilde{\mathscr{R}}^X$, $g_2 \in \tilde{\mathscr{R}}^X$, and $g_3 \in \tilde{\mathscr{R}}^X$. Note how the predecessor vector of node 2 is in the sub-threshold predecessor set of X even if node 2 is in X. Note how node 3 has one missing predecessor (i.e., one of the elements in g_3 is $\perp$). The nodes whose predecessor vectors are in $\tilde{\mathscr{R}}^X$ are highlighted.

Let $g = (v, e)$ be the multigraph on which Threshold Contagion is played. At the beginning of round r, the player knows the value of $W_i^{r'}[j]$ for every $r' < r$, every $i \in \mathbb{N}$ and every $j \in v$, which we encode in an *infection history*. The player chooses a set of S of the nodes that are healthy at the beginning of round r. We model this choice with an *infection function*. We call *infection strategy* the sequence of choices the player makes throughout the game.

Definition 42 (Infection history)**.** An **infection history** for round $r > 0$ is an element of the set

$$\mathscr{H}_r = \left(\left(\{\perp, \top\}^N \right)^\infty \right)^r$$

An infection history is a table with three indices. The first represents the round, the second represents the step, the third represents the node. Let $h \in \mathscr{H}_r$, then $h_{i'}^{r'}[j] = \top$ signifies that node j is infected at round r' and step i'.

Notation 13 (Round and step order)**.** Let r, r' be round numbers, let i, i' be step numbers. We say that $(r, i) < (r', i')$ if (r, i) *temporally precedes* (r', i'). More formally

$$(r, i) < (r', i') \iff \left(r' > r\right) \vee (r' = r \wedge i' > i)$$

Definition 43 (Valid infection history)**.** A **valid infection history** for round $r > 0$ is an element of the set

$$\mathscr{H}_r^* = \left\{ h \in \mathscr{H}_r \mid h_{i'}^{r'}[j] = \top \implies h_{i''}^{r''}[j] = \top \; \forall (r'', i'') > (r', i') \right\}$$

A valid infection history is an infection history where a node is never healed. If a node is

infected at round r' and step i', then it also infected at any subsequent round r'' and step i''.

Definition 44 (Incomplete infection history)**.** An **incomplete infection history** for round $r > 0$ is an element of the set

$$\mathscr{H}_r^+ = \left\{h \in \mathscr{H}_r^* \mid \left|\left\{j \in 1..N \mid h_\infty^{r-1}[j] = \bot\right\}\right| \geq S\right\}$$

An incomplete infection history is a valid infection history with at least S healthy nodes at the end of round $r-1$.

Definition 45 (Infection function)**.** An **infection function** for round r is an element of the set

$$\mathscr{F}_r = \left\{f : \mathscr{H}_r^+ \to \mathbb{P}^S(\{1..N\}) \mid \forall x \in f(h), h_\infty^{r-1}[x] = \bot\right\}$$

An infection function is a function that inputs an incomplete infection history and outputs a set of S nodes, all of which are healthy at the end of round $r-1$.

Definition 46 (Infection strategy)**.** An **infection strategy** is an element of the set

$$\mathscr{F} = \mathbb{P}^{1..N}(S) \times \prod_{r=1}^{R-1} \mathscr{F}_r$$

The first element of an infection strategy is a set of S nodes to infect at the beginning of round 0. Let $r > 0$, the r-th element of an infection strategy is an infection function for round r.

An infection strategy encodes all the choices a player makes during a game of Threshold Contagion:

- At the beginning of round 0, the player has no information available. All nodes are healthy, and its choice reduces to selecting S of them to infect.
- At the beginning of round $r \geq 1$, the information available to the player is the propagation of the infection throughout all previous rounds. Such information is input to the r-th infection function, which returns a set of S healthy nodes to infect.

Sample space

In Section 4.11.6, we noticed how the outcome of a game of Threshold Contagion is completely determined once both the topology of the multigraph and the strategy of the player are known.

In Section 4.11.6, we showed how a multigraph can be expressed with a predecessor matrix, defined the space of predecessor matrices and derived the probability distribution underlying random predecessor matrices.

In Section 4.11.6, we showed how the choices that a player makes at the beginning of each round in response to the infection history can be encoded in infection strategies. We then defined the space of infection strategies. Unlike random multigraphs, infection strategies are under the control of the player. Therefore, a probability distribution over the space of infection strategies is not available.

As we discussed in Section 4.11.6, an element of the sample space is a pair of a multigraph and an infection strategy.

Definition 47 (Sample space)**.** The **sample space** for Threshold Contagion is the set $\Omega = \mathscr{G} \times \mathscr{F}$.

Lemma 43. *Let* $\omega = (g, f)$ *be a random element of* Ω. *Then* $\mathscr{P}[\bar{g}, \bar{f}] = \mathscr{P}[\bar{g}]\mathscr{P}[\bar{f}]$, *i.e., g and f are independent.*

Proof. It immediately follows from the fact that the player has no knowledge of the topology of the multigraph g. □

4.11.7 Random variables as sample functions

In Section 4.11.4 we intuitively defined a set of random variables to capture useful properties of a game of Threshold Contagion. In the next sections, we use those random variables to compute the probability distribution underlying the number of infected nodes at the end of a game.

In Section 4.11.6 we formally defined the sample space of a game of Threshold Contagion. We started by showing that an instance of the game is completely determined once the topology of the multigraph and the strategy of the player are known. We also computed the probability of any specific multigraph topology occurring.

In this section, we rigorously re-define the random variables we defined in Section 4.11.4 by expressing them as functions on the sample space as defined in Section 4.11.6.

Infection history

As discussed in Section 4.11.6, an infection function for round r inputs an incomplete infection history for round r and outputs a set of S nodes to infect out of those that are healthy at the end of round $r-1$.

We introduce two useful functions to manipulate infection histories.

Definition 48 (Sample history function)**.** The **sample history function** for round r is the function $\mathfrak{h}_r : \Omega \to \mathscr{H}_r^*$ defined by

$$(\mathfrak{h}_r(\omega))_i^{r'}[j] = W_i^{r'}[j](\omega)$$

The sample history function for round r inputs a sample ω and outputs the valid infection history for round r produced by ω.

Note how the definition of sample history function relies on the definition of the infection status W. We introduced W in Section 4.11.4, and we formally define it in the next section.

Definition 49 (Sample completion function)**.** The **sample completion function** for round r is the function $\mathfrak{c}_r : \Omega \to \{\top, \bot\}$ defined by

$$\mathfrak{c}_r(\omega) = \begin{cases} \bot & \text{iff } \mathfrak{h}_r(\omega) \in \mathscr{H}_r^+ \\ \top & \text{otherwise} \end{cases}$$

The sample completion function for round r inputs a sample and outputs $\top$ if the infection history of the sample is complete at round r, and $\bot$ otherwise.

Infection status

As stated in Section 4.11.3, the infection status is defined as follows:

- At the beginning of the game, all the nodes are healthy.
- During the first step of each round, the player selects a set of S healthy nodes and infects them.
- During every subsequent step, every healthy node that has at least $\hat{R}$ infected predecessors is infected.
- The infection state at the end of a round is carried without change to the beginning of the next round.

In order to formalize the above in the definition of infection status, we preliminarly define *infection sets.*

Definition 50 (Infection set)**.** The **infection set** at round r and step i is the random variable $\hat{W}_i^r : \Omega \to \mathbb{P}(1..N)$ defined by

$$\hat{W}_i^r(\omega) = \left\{ j \in 1..N \mid W_i^r[j](\omega) = \top \right\}$$

The infection set $\hat{W}_i^r(\omega)$ represents the set of nodes that are infected in ω at round r and step i.

Like the sample history function, the definition of infection set relies on the definition of infection status W, which we can now define by cases.

Definition 51 (Infection status). Let $\omega = (g, f) \in \Omega$. The **infection status** for round r, step i and node j is the random variable $W_i^r[j] : \Omega \to \{\top, \bot\}$ defined by

$$W_0^0[j](\omega) = \bot \tag{4.5}$$

$$W_0^{r>0}[j](\omega) = W_\infty^{r-1}[j](\omega) \tag{4.6}$$

$$W_1^0[j](\omega) = \begin{cases} \top & \text{iff } j \in f_0 \\ W_0^0[j](\omega) & \text{otherwise} \end{cases} \tag{4.7}$$

$$W_1^{r>0}[j](\omega) = \begin{cases} \top & \text{iff } \mathfrak{c}_r(\omega) = \bot \wedge j \in f_r(\mathfrak{h}_r(\omega)) \\ W_0^r[j](\omega) & \text{otherwise} \end{cases} \tag{4.8}$$

$$W_{i>1}^r[j](\omega) = \begin{cases} W_{i-1}^r[j](\omega) & \text{iff } g_j \in \tilde{\mathscr{R}}^{\hat{W}_{i-1}^r(\omega)} \\ \top & \text{otherwise} \end{cases} \tag{4.9}$$

The above equations encode the following properties:

- At the beginning of the game (Equation (4.5)), all nodes are healthy.
- The infection status at the beginning of round $r > 0$ (Equation (4.6)) is equal to the infection status at the end of round $r - 1$.
- During step 1 of round 0 (Equation (4.7)), all the nodes in f_0 are infected. Intuitively, the player selects S nodes and infects them. Note how this choice is not informed by any history (following from Definition 46, f_0 is a set and not a function).
- During step 1 of round $r > 0$ (Equation (4.8)), if ω is not complete (i.e., there are at least S healthy nodes at the beginning of round r), all the nodes in $f_r(\mathfrak{h}_r(\omega))$ are infected. Intuitively, the player selects S healthy nodes and infects them. This choice is informed by the infection history for round r (see Definition 48).
- During step $i > 0$ of any round r (Equation (4.9)), all the nodes whose predecessor vector is not in the sub-threshold predecessor set (see Definition 41) of the infection set at step $i - 1$ are infected. In other words, the contagion rule (see Section 4.11.1) is applied, and all the nodes that have at least $\hat{R}$ infected predecessors are infected.

Following from Definition 51, we prove that nodes are never healed in a game of Threshold Contagion.

Lemma 44. *Let $j \in 1..N$, $r, r' \in 1..K$, $i, i' \in \mathbb{N}$, let $\omega \in \Omega$. If $(r', i') \geq (r, i)$, then*

$$W_i^r[j](\omega) = \top \implies W_{i'}^{r'}[j](\omega)$$

Proof. Let $r'' \in 1..K$, $i'' \in \mathbb{N}$. Following from Equations (4.5) to (4.9), we have

$$W_{i''+1}^{r''}[j](\omega) \neq W_{i''}^{r''}[j](\omega) \implies W_{i''+1}^{r''}[j](\omega) = \top \tag{4.10}$$

The lemma is proved by induction on Equations (4.6) and (4.10). □

Corollary 3. *The infection set $\hat{W}_i^r(\omega)$ is non-decreasing in (r, i).*

Infection size, frontier size and infected predecessors count

In Section 4.11.7, we defined the infection status $W_i^r[j]$ as a function on the sample space (see Definition 51). We also defined the infection set $\hat{W}_i^r$ as the set of nodes for which $W_i^r = \top$ (see Definition 50).

As stated in Section 4.11.4, the infection size N_i^r represents the number of infected nodes at round r and step i, and the frontier size $U_{i>0}^r$ represents the number of nodes that are infected at round r and step i, but not at step $i-1$. We can formalize the above in the following definitions.

Definition 52 (Infection size)**.** The **infection size** for round r and step i is the random variable $N_i^r : \Omega \to 0..N$ defined by

$$N_i^r(\omega) = \left|\hat{W}_i^r(\omega)\right|$$

The infection size counts the infected nodes at step (r, i).

Definition 53 (Frontier size)**.** The **frontier size** for round r and step i is the random variable $U_i^r : \Omega \to 0..N$ defined by

$$U_{i>0}^r(\omega) = N_i^r(\omega) - N_{i-1}^r(\omega)$$

The infection size counts the nodes that are infected at step (r, i), but not at step $(r, i-1)$.

As stated in Section 4.11.4, the infected predecessors count of node j for round r and step i represents the number of predecessors of node j that are infected at round r and step i. We can formalize this definition in the following.

Definition 54 (Infected predecessors count)**.** Let $\omega = (g, f) \in \Omega$. The **infected predecessors count** of node j for round r and step i is the random variable $V_i^r[j] : \Omega \to 0..R$ defined by

$$V_i^r[j](\omega) = \left|\left\{k \mid \left(g_{j,k}\right) \in \hat{W}_i^r(\omega)\right\}\right|$$

The infected predecessors count counts the number of predecessors of node j that are infected at step (r, i).

Lemma 45. *Let $\omega = (g, f) \in \Omega$, let $j \in 1..N$, $r \in 1..K$, $i \in \mathbb{N}$. Then*

$$g \in \tilde{\mathscr{R}}^{\hat{W}_i^r[j]} \iff V_i^r[j](\omega) \le \hat{R}$$

Proof. It follows immediately from Definitions 41 and 54. □

4.11.8 Contagion step

In Section 4.11.7, we expressed the random variables we introduced in Section 4.11.4 as functions over the elements of the sample space we defined in Section 4.11.6. As we established in Section 4.11.5, the goal of this section is to compute the distribution underlying N_∞^K (see Equation (4.3)).

Here we focus on the contagion steps of a round of Threshold Contagion. As per Equation (4.9), at every step (r, i) such that $i > 1$, all the healthy nodes that have at least $\hat{R}$ infected predecessors become infected.

In this section, we show that a contagion step defines a Markov chain with states $(\bar{N}_i^r, \bar{U}_i^r)$. More formally, we show that a transition matrix $\mathscr{M}$ exists such that, for every $(\bar{N}, \bar{U})$, $(\bar{N}', \bar{U}')$ and for every $r \in 1..K$, $i \geq 1$,

$$\mathscr{M}_{\bar{N},\bar{U}}^{\bar{N}',\bar{U}'} = \mathscr{P}\left[N_{i+1}^r = \bar{N}', U_{i+1}^r = \bar{U}' \mid N_i^r = \bar{N}, U_i^r = \bar{U}\right] \tag{4.11}$$

Intuitively, this means that, once the infection size and the frontier size at step (r, i) are determined, no other knowledge is needed to compute the probability distribution underlying the frontier size at step $(r, i+1)$. This means, in particular, that the player's infection strategy does not affect the end result of the game. This result is somewhat unsurprising: since the player has no knowledge of the multigraph on which Threshold Contagion is played, the player has no way to meaningfully distinguish two nodes by the effect that their infection will have on the system. Since the number of infected nodes per round is determined, every choice of the player can be shown to be effectively equivalent to the infection of S random healthy nodes.

Roadmap

Notation 14 (Markov states)**.** We use $\langle \bar{N}_i^r, \bar{U}_i^r \rangle$ to denote the subset of the sample space Ω that satisfies $N_i^r(\omega \in \Omega) = \bar{N}_i^r, U_i^r(\omega \in \Omega) = \bar{U}_i^r$.

Equivalently,

$$\langle \bar{N}_i^r, \bar{U}_i^r \rangle = \left(N_i^r\right)^{-1}\left(\bar{N}_i^r\right) \cap \left(U_i^r\right)^{-1}\left(\bar{U}_i^r\right)$$

In order to show that a infection step defines a Markov chain with states $(\bar{N}_i^r, \bar{U}_i^r)$, we:

- Define a set of *partition functions* $\mathscr{S}_i^r : \Omega \to \mathbb{P}(\Omega)$ that map elements of Ω into well-structured subsets of Ω. Intuitively, $\mathscr{S}_i^r$ maps a sample $\omega = (g, f)$ to a set of samples that are *similar to it* (by a notion of similarity that we define later).
- Let $\omega' \in \mathscr{S}_i^r(\omega)$. We show that ω and ω' result in the same infection history up to step (r, i).

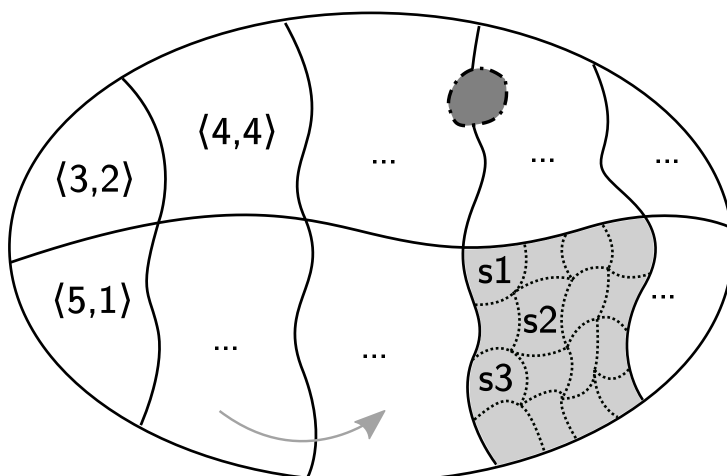


Figure 4.3: An illustration of sample space and the steps needed to show that a contagion step defines a Markov chain. The grey arrow represents a transition from a state to another. One of the states is further partitioned by $\mathscr{S}_i^r$. The dark grey area represents a case that we prove won't happen.

- We show that $\mathscr{S}_i^r$ can be used to define an equivalence relation on the sample space Ω.
- Let ω be equivalent to ω' through $\mathscr{S}_i^r$. We show that, since $N_i^r(\omega) = N_i^r(\omega')$ and $U_i^r(\omega) = U_i^r(\omega')$, then $\mathscr{S}_i^r$ can be used to quotient $\langle \bar{N}_i^r, \bar{U}_i^r \rangle$.
- Let $r \in 1..K$, $i > 1$. We use $\mathscr{S}_i^r$ to partition $\langle \bar{N}_i^r, \bar{U}_i^r \rangle$ in $s_1, \ldots, s_q$. We show that the probability of ω being in $\langle \bar{N}_{i+1}^r, \bar{U}_{i+1}^r \rangle$ given that ω is in s_h is analitically computable and independent of h.
- We use the independence across partitions to compute the terms of $\mathscr{M}_{\bar{N},\bar{U}}^{\bar{N}',\bar{U}'}$

Partition functions

We start by defining a set of *partition functions* $\mathscr{S}_i^r : \Omega \to \mathbb{P}(\Omega)$ that map elements of Ω into subsets of Ω. Intuitively, a partition function maps a sample to a set of samples that are *similar* to it.

Let $\omega = (g, f) \in \Omega$, let $\omega' = (g', f') \in \mathscr{S}_i^r(\omega)$. We define $\mathscr{S}_i^r$ such that the following hold:

- $f' = f$, i.e., the player's strategy is left unchanged by $\mathscr{S}_i^r$.
- Let $j \in 1..N$ be a node. If j is infected in ω at step (r, i), then $g'_j = g_j$. In other words, the predecessors of a node that is infected at step (r, i) in ω are left unchanged by $\mathscr{S}_i^r$.
- Let $j \in 1..N$ be a node. If j is not infected in ω at step (r, i), then g'_j is an element of the sub-threshold predecessor set of $\hat{W}_{i-1}^r(\omega)$. In other words, the predecessors of a node

that is not infected at step (r,i) in ω can be changed by $\mathscr{S}_i^r$, as long as no more than $\hat{R}$ of them are infected in ω' at step $(r,i-1)$. Intuitively, we allow the predecessors of j to change in a way that does not make it infected in ω at step (r,i).

We formalize the above in the following definition.

Definition 55 (Partition function)**.** Let $r \in 1..K$, $i \geq 1$, let $\omega = (g,f) \in \Omega$. The **partition function** for round r and step i is the function $\mathscr{S}_i^r : \Omega \to \mathbb{P}(\Omega)$ defined by

$$\mathscr{S}_i^r(\omega) = \left(\prod_{j=1}^{N} \mathscr{S}_i^r[j](\omega)\right) \times \{f\} \tag{4.12}$$

$$\mathscr{S}_i^r[j](\omega) = \begin{cases} \{g_j\} & \text{iff } W_i^r[j](\omega) = \top \\ \tilde{R}^{\hat{W}_{i-1}^r(\omega)} & \text{otherwise} \end{cases} \tag{4.13}$$

Infection history

In Section 4.11.8 we defined a set of partitions functions that map a sample $\omega \in \Omega$ to a set of samples that are *similar* to ω.

Let $\omega = (g,f) \in \Omega$. We designed $\mathscr{S}_i^r$ to leave unchanged the player's strategy and the predecessors of every node that is infected in ω at step (r,i). The predecessors of the nodes that are not infected in ω at step (r,i) can change, as long as less than $\hat{R}$ of them are among the nodes that are infected in ω at step $(r,i-1)$.

Intuitively, $\mathscr{S}_i^r$ is designed to alter the topology of g in a way that does not affect its infection history: since the predecessors of the nodes that are not infected in ω at step (r,i) are not changed, they will still be infected in ω'. Similarly, if a node is not infected in ω at step (r,i), its predecessors are not changed in a way that makes it infected in ω' at step (r,i).

In this section, we formally prove this intuitive result.

Lemma 46. *Let $j \in 1..N$, let $\omega, \omega' \in \Omega$. If $\omega' \in \mathscr{S}_i^r(\omega)$, then for every $(r',i') \leq (r,i)$*

$$W_{i'}^{r'}[j](\omega') = W_{i'}^{r'}[j](\omega)$$

Proof. Let $\omega = (g,f)$ and $\omega' = (g',f')$. We prove the lemma by induction. We start by noting that, following from Equation (4.5),

$$W_0^0[j](\omega') = \bot = W_0^0[j](\omega)$$

Now, assume that $(r',i') < (r,i)$ and, for all $j \in 1..N$, $\hat{W}_{i'}^{r'}[j](\omega') = \hat{W}_{i'}^{r'}[j](\omega)$.

If $r' = 0$ and $i' = 0$, then from Equation (4.7) it follows that, if $j \in \left(f_0 = f_0'\right)$,

$$W_1^0[j](\omega') = \top = W_1^0[j](\omega)$$

and, otherwise,

$$W_1^0[j](\omega') = W_0^0[j](\omega') = W_0^0[j](\omega) = W_1^0[j](\omega)$$

If $r' > 0$ and $i' = 0$, then $\mathfrak{h}_r(\omega') = \mathfrak{h}_r(\omega)$. Following from Equation (4.8), if $\left(\mathfrak{c}_r(\omega) = \mathfrak{c}_r(\omega')\right) = \bot$ and $j \in \left(f_r(\mathfrak{h}_r(\omega)) = f_r'(\mathfrak{h}_r(\omega'))\right)$, then

$$W_1^{r'}[j](\omega') = \top = W_1^{r'}[j](\omega)$$

and otherwise

$$W_1^{r'}[j](\omega') = W_0^{r'}[j](\omega') = W_0^{r'}[j](\omega) = W_1^{r'}[j](\omega)$$

We now consider the case $i' \geq 1$. We start by noting that, since $\hat{W}_{i'}^{r'}(\omega') = \hat{W}_{i'}^{r'}(\omega)$, then $\tilde{\mathscr{R}}^{\hat{W}_{i'}^{r'}(\omega')} = \tilde{\mathscr{R}}^{\hat{W}_{i'}^{r'}(\omega)}$.

If $W_i^r[j](\omega) = \top$, then $g_j = g_j'$. Following from Equation (4.9), if $\left(g_j' = g_j\right) \in \left(\tilde{\mathscr{R}}^{\hat{W}_{i'}^{r'}(\omega')} = \tilde{\mathscr{R}}^{\hat{W}_{i'}^{r'}(\omega)}\right)$, then

$$W_{i'+1}^{r'}[j](\omega') = W_{i'}^{r'}[j](\omega') = W_{i'}^{r'}[j](\omega) = W_{i'+1}^{r'}[j](\omega)$$

and otherwise

$$W_{i'+1}^{r'}[j](\omega') = \top = W_{i'+1}^{r'}[j](\omega)$$

If $W_i^r[j](\omega) = \bot$, then $g_j' \in \tilde{\mathscr{R}}^{\hat{W}_{i-1}^r(\omega)}$. Noting that $(r', i') \leq (r, i-1)$, from Lemma 44 it follows that $\hat{W}_{i'}^{r'}(\omega) \subseteq \hat{W}_{i-1}^r(\omega)$, and consequently, from Equation (4.9), we have

$$g_j' \in \left(\tilde{\mathscr{R}}^{\hat{W}_{i-1}^r(\omega)} \subseteq \tilde{\mathscr{R}}^{\hat{W}_{i'}^{r'}(\omega)} = \tilde{\mathscr{R}}^{\hat{W}_{i'}^{r'}(\omega')}\right)$$

Moreover, since $W_i^r[j](\omega) = \bot$, from Lemma 44 it follows

$$W_{i'+1}^{r'}(\omega) = W_{i'}^{r'}(\omega) = W_{i'}^{r'}(\omega') = \bot$$

and therefore

$$W_{i'+1}^{r'}(\omega') = W_{i'}^{r'}(\omega') = W_{i'+1}^{r'}(\omega)$$

Finally, if $i = \infty$, then following from Equation (4.6) we have

$$W_0^{r'+1}[j](\omega') = W_\infty^{r'}[j](\omega') = W_\infty^{r'}[j](\omega) = W_0^{r'+1}[j](\omega)$$

□

Corollary 4. *Let* $\omega, \omega' \in \Omega$. *If* $\omega' \in \mathscr{S}_i^r(\omega)$, *then*

$$N_i^r(\omega') = N_i^r(\omega)$$
$$U_i^r(\omega') = U_i^r(\omega)$$

Equivalence relation

In Section 4.11.8, we introduced a set of functions $\mathscr{S}_i^r : \Omega \to \mathbb{P}(\Omega)$ that map a sample into a set of *similar* samples. In Section 4.11.8, we proved that, if $\omega \in \Omega$ and $\omega' \in \mathscr{S}_i^r(\omega)$, then ω and ω' produce the same infection history (i.e., the same values for $\hat{W}_i^r$) up to round r and step i.

In this section, we show that $\mathscr{S}_i^r$ can be used to define a equivalence relation on Ω.

Lemma 47. *Let* $\omega, \omega' \in \Omega$. *If* $\omega' \in \mathscr{S}_i^r(\omega)$, *then* $\mathscr{S}_i^r(\omega') = \mathscr{S}_i^r(\omega)$.

Proof. Let $\omega = (g, f)$, $\omega' = (g', f')$. Following from Lemma 46, for every j we have

$$W_i^r[j](\omega') = W_i^r[j](\omega)$$
$$W_{i-1}^r[j](\omega') = W_{i-1}^r[j](\omega)$$

Following from Definition 55, if $W_i^r[j](\omega) = \top$, then $g'_j = g_j$. Consequently,

$$\mathscr{S}_i^r[j](\omega') = \left\{g'_j\right\} = \{g_j\} = \mathscr{S}_i^r[j](\omega)$$

If $W_i^r[j](\omega) = \perp$, then

$$\mathscr{S}_i^r[j](\omega') = \tilde{\mathscr{R}}^{\hat{W}_{i-1}^r[j](\omega')} = \tilde{\mathscr{R}}^{\hat{W}_{i-1}^r[j](\omega)} = \mathscr{S}_i^r[j](\omega)$$

Therefore,

$$\mathscr{S}_i^r(\omega') = \prod_{j=1}^{N} \mathscr{S}_i^r[j](\omega') = \prod_{j=1}^{N} \mathscr{S}_i^r[j](\omega) = \mathscr{S}_i^r(\omega)$$

□

Definition 56 (Partition relation)**.** Let $\omega, \omega' \in \Omega$. If $\omega' \in \mathscr{S}_i^r(\omega)$, then ω' has a **partition relation** with ω at round r and step i:

$$\omega' \overset{(r,i)}{\sim} \omega$$

Lemma 48. $\overset{(r,i)}{\sim}$ *is an equivalence relation.*

Proof. Let $j \in 1..N$, let $\omega \in \Omega$. Following from Equation (4.9), if $W_i^r[j](\omega) = \bot$, then $g_j \in \tilde{\mathscr{R}}^{\hat{W}_{i-1}^r(\omega)}$. Consequently, following from Definition 55, if $W_i^r[j](\omega) = \top$, then

$$g_j \in \{g_j\} = \mathscr{S}_i^r[j](\omega)$$

and if $W_i^r[j](\omega) = \bot$, then

$$g_j \in \tilde{\mathscr{R}}^{\hat{W}_{i-1}^r[j](\omega)} = \mathscr{S}_i^r[j](\omega)$$

Therefore, $\omega \in \mathscr{S}_i^r(\omega)$, and

$$\omega \overset{(r,i)}{\sim} \omega$$

therefore $\overset{(r,i)}{\sim}$ is reflexive.

Let $\omega' \in \mathscr{S}_i^r(\omega)$. By Lemma 47, $\mathscr{S}_i^r(\omega') = \mathscr{S}_i^r(\omega)$. Consequently

$$\omega \in \left(\mathscr{S}_i^r(\omega) = \mathscr{S}_i^r(\omega')\right)$$

and

$$\omega' \overset{(r,i)}{\sim} \omega \implies \omega \overset{(r,i)}{\sim} \omega'$$

therefore $\overset{(r,i)}{\sim}$ is symmetric.

Let $\omega'' \in \mathscr{S}_i^r(\omega')$. Again by Lemma 47,

$$\omega'' \in \left(\mathscr{S}_i^r(\omega') = \mathscr{S}_i^r(\omega)\right)$$

and

$$\omega' \overset{(r,i)}{\sim} \omega, \omega'' \overset{(r,i)}{\sim} \omega' \implies \omega'' \overset{(r,i)}{\sim} \omega$$

therefore, $\overset{(r,i)}{\sim}$ is transitive. □

Transition probabilities

In Section 4.11.8 we showed that the partition function we introduced in Section 4.11.8 can be used to induce an equivalence relation on the sample space Ω.

In this section, we use this result to show that a contagion step defines a Markov chain with states $(\bar{N}_i^r, \bar{U}_i^r)$, and compute the values of its associated transition matrix $\mathscr{M}$.

More formally, let $r \in 1..K$, $i \geq 1$. In this section, we compute

$$\mathscr{P}[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid \bar{N}_i^r, \bar{U}_i^r]$$

and we show that its value is independent of the player's strategy.

As we established in Lemma 48, $\overset{(r,i)}{\sim}$ is an equivalence relation on Ω. Moreover, let $\omega \in \Omega$, by

Corollary 4 we have $\mathscr{S}_i^r(\omega) \subseteq \langle \bar{N}_i^r, \bar{U}_i^r \rangle$.

We can therefore use $\overset{(r,i)}{\sim}$ to partition $\langle \bar{N}_i^r, \bar{U}_i^r \rangle$:

$$\{s_1, \dots, s_q\} = \frac{\langle \bar{N}_i^r, \bar{U}_i^r \rangle}{\overset{(r,i)}{\sim}}$$

By the law of total probability,

$$\begin{aligned} \mathscr{P}[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid \bar{N}_i^r, \bar{U}_i^r] &= \mathscr{P}[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid \langle \bar{N}_i^r, \bar{U}_i^r \rangle] \\ &= \sum_{l=1}^{q} \mathscr{P}[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid s_l] \mathscr{P}[s_l \mid \langle \bar{N}_i^r, \bar{U}_i^r \rangle] \end{aligned}$$

Note how $\mathscr{P}[s_l \mid \langle \bar{N}_i^r, \bar{U}_i^r \rangle]$ is unknown, as it depends on the probability distribution underlying the player's strategy. For a given h, we instead focus on computing $\mathscr{P}[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid s_h]$.

Roadmap. In order to compute $\mathscr{P}[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid s_h]$, we compute the probability for a node that is not infected in s_h at step (r, i) to become infected at time $(r, i+1)$. Let j be a node that is not infected in s_h at step (r, i). We compute the probability of it becoming infected at step $(r, i+1)$ by first computing the probability distribution underlying $V_{i-1}^r[j]$. Given $\bar{V}_{i-1}^r[j]$, we then compute the probability distribution underlying $V_i^r[j]$, and threshold it with $\hat{R}$ to compute the probability of j becoming infected at step $i+1$.

Notation 15 (Kronecker delta)**.** We use δ to denote the **Kronecker delta**. Let $i, j \in \mathbb{N}$, then

$$\delta_{i,j} = I(i = j)$$

Let $\bar{\omega} = (\bar{g}, \bar{f}) \in s_h$ be an example of s_h. Let $W, \not{W}$ denote the set of nodes that are infected and not infected in $\bar{\omega}$ at step (r, i), respectively:

$$\begin{aligned} W = \{w_1, \dots, w_n\} &= \hat{W}_r^i(\bar{\omega}) \\ \not{W} = \{\not{w}_1, \dots, \not{w}_m\} &= 1..N \setminus \hat{W}_r^i(\bar{\omega}) \end{aligned}$$

with $n = N_i^r(\bar{\omega})$ and $m = N - n$. Let $\omega = (g, f)$, following from Lemma 45 we have

$$(\omega \in s_h) \Longleftrightarrow \left(g_{w_1} = \bar{g}_{w_1}, \dots, g_{w_n} = \bar{g}_{w_n}, V_{i-1}^r[\not{w}_1] \leq \hat{R}, \dots, V_{i-1}^r[\not{w}_m] \leq \hat{R}\right)$$

Let $j \in \not{W}$, i.e., $W_i^r[j] = \bot$. Using the independence of the distribution of each predecessor vector in s_h (see Equation (4.4) and Definition 55), we can compute the probability distribution

underlying $V_{i-1}^r[j]$ in s_h:

$$
\begin{aligned}
&\mathscr{P}\left[\bar{V}_{i-1}^r[j] \mid \cancel{W_i^r[j]}, s_h\right] \\
= \;& \mathscr{P}\left[\bar{V}_{i-1}^r[j] \mid \cancel{W_i^r[j]}, \bar{g}_{w_1}, \ldots, \bar{g}_{w_n}, r_{i-1}[w_1] < \hat{R}, \ldots, V_{i-1}^r[w_m] < \hat{R}\right] \\
= \;& \mathscr{P}\left[\bar{V}_{i-1}^r[j] \mid \cancel{W_i^r[j]}, V_{i-1}^r[w_1] < \hat{R}, \ldots, V_{i-1}^r[w_m] < \hat{R}\right] \\
= \;& \mathscr{P}\left[\bar{V}_{i-1}^r[j] \mid V_{i-1}^r[j] < \hat{R}\right]
\end{aligned}
$$

Using Bayes' theorem we get

$$
\mathscr{P}\left[\bar{V}_{i-1}^r \mid V_{i-1}^r < \hat{R}\right] = \frac{\mathscr{P}\left[V_{i-1}^r < \hat{R} \mid \bar{V}_{i-1}^r\right]\mathscr{P}\left[\bar{V}_{i-1}^r\right]}{\mathscr{P}\left[V_{i-1}^r < \hat{R}\right]} \tag{4.14}
$$

Following from Lemma 41, each predecessor of j is independently selected with uniform probability. Given $\bar{N}_{i-1}^r$, each predecessor of j has a probability $l\left(\bar{N}_{i-1}^r / N\right)$ of being in $\hat{W}_{i-1}^r$. The unconditioned number of infected predecessors of j is therefore binomially distributed:

$$
\mathscr{P}\left[\bar{V}_{i-1}^r\right] = \mathrm{Bin}\left[E, l\frac{\bar{N}_{i-1}^r}{N}\right]\left(\bar{V}_{i-1}^r\right) \tag{4.15}
$$

Plugging Equation (4.15) in Equation (4.14) and noting that $N_{i-1}^r = N_i^r - U_i^r$ we get

$$
\mathscr{P}\left[\bar{V}_{i-1}^r \mid V_{i-1}^r < \hat{R}\right] = \frac{I\left(\bar{V}_{i-1}^r < \hat{R}\right)\mathrm{Bin}\left[R, l\frac{\bar{N}_i^r - \bar{U}_i^r}{N}\right]\left(\bar{V}_{i-1}^r\right)}{\sum_{\bar{V}=0}^{\hat{R}-1}\mathrm{Bin}\left[R, l\frac{\bar{N}_i^r - \bar{U}_i^r}{N}\right]\left(\bar{V}\right)}
$$

We now compute the distribution underlying V_i^r, given $\bar{V}_{i-1}^r$, $\cancel{W_i^r}$ and s_h. Given $\bar{V}_{i-1}^r$, $\cancel{W_i^r}$ and s_h, j has $E - \bar{V}_{i-1}^r$ predecessors that are not in $\hat{W}_{i-1}^r$. Let $g_{j,k}$ be a predecessor of j that is not in $\hat{W}_{i-1}^r$, we have

$$
\begin{aligned}
\mathscr{P}\left[g_{j,k} \in \hat{W}_i^r \mid g_{j,k} \notin \hat{W}_{i-1}^r\right] &= \frac{\mathscr{P}\left[g_{j,k} \in \hat{W}_i^r, g_{j,k} \notin \hat{W}_{i-1}^r\right]}{\mathscr{P}\left[g_{j,k} \notin \hat{W}_{i-1}^r\right]} \\
&= \frac{l\frac{\bar{U}_i}{N}}{1 - l\frac{\bar{N}_i^r - \bar{U}_i^r}{N}}
\end{aligned}
$$

Following from Equation (4.4), each predecessor of j that is not in $\hat{W}_{i-1}^r$ has an independent chance of being in $\hat{W}_i^r$. Therefore, the number of newly infected predecessors for j at step i is

binomially distributed:

$$\mathscr{P}\left[\bar{V}_i^r \mid \bar{V}_{i-1}^r, \not{W}_i^r, s_h\right] = \text{Bin}\left[R - \bar{V}_{i-1}^r, \frac{l\frac{\bar{U}_i}{N}}{1 - l\frac{\bar{N}_i^r - \bar{U}_i^r}{N}}\right]\left(\bar{V}_i^r - \bar{V}_{i-1}^r\right) \tag{4.16}$$

Using the law of total probability, we can now use Equations (4.14) and (4.16) to compute the probability distribution underlying $V_i^r[j]$, given $\not{W}_i^r$ and s_h:

$$\mathscr{P}\left[\bar{V}_i^r \mid \not{W}_i^r, s_h\right] = \sum_{\bar{V}_{i-1}^r=0}^{\hat{R}-1} \mathscr{P}\left[\bar{V}_i^r \mid \bar{V}_{i-1}^r, \not{W}_i^r, s_h\right]\mathscr{P}\left[\bar{V}_{i-1}^r \mid \not{W}_i^r, s_h\right]$$

Finally, following from Lemma 45, we get the probability of $W_i^r[j]$, given $\not{W}_i^r$ and s_h:

$$\mathscr{P}\left[W_i^r \mid \not{W}_i^r, s_h\right] = \sum_{\bar{V}_i^r=\hat{R}}^{R} \mathscr{P}\left[\bar{V}_i^r \mid \not{W}_i^r, s_h\right]$$

Since each of the $N - \bar{N}_i$ nodes in $\not{W}$ has an independent probability of becoming infected at round $i+1$, the frontier size at step $i+1$, given s_h is binomially distributed:

$$\mathscr{P}\left[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid s_h\right] = \text{Bin}\left[N - \bar{N}_i, \mathscr{P}\left[W_{i+1}^r \mid \not{W}_i^r, s_h\right]\right]\left(\bar{U}_{i+1}^r\right)\delta_{\bar{N}_{i+1}^r - \bar{N}_i^r, \bar{U}_{i+1}^r}$$

We can now note how, when computing $\mathscr{P}\left[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid s_h\right]$, the condition on s_h reduces only to a condition on the values of $\bar{N}_i^r$ and $\bar{U}_i^r$. Since $s_1, \ldots, s_q$ share the same values of $(\bar{N}_i^r, \bar{U}_i^r)$, the transition probability for the Markov chain underlying a contagion step reduces to

$$\begin{aligned}\mathscr{P}\left[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid \bar{N}_i^r, \bar{U}_i^r\right] &= \sum_{l=1}^{q} \mathscr{P}\left[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid s_l\right]\mathscr{P}\left[s_l \mid \langle \bar{N}_i^r, \bar{U}_i^r \rangle\right] \\ &= \mathscr{P}\left[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid s_h\right]\sum_{l=1}^{q} \mathscr{P}\left[s_l \mid \langle \bar{N}_i^r, \bar{U}_i^r \rangle\right] \\ &= \mathscr{P}\left[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r \mid s_h\right]\end{aligned} \tag{4.17}$$

4.11.9 Final infection size

In Section 4.11.8, we showed that a contagion steps defines a Markov chain with states $(\bar{N}_i^r, \bar{U}_i^r)$, and we computed the values of its associated transition matrix $\mathscr{M}$. In this section, we use this result to achieve our goal to compute the probability distribution underlying the infection size at the end of a game of Threshold Contagion.

As we established in Section 4.11.8, provided with $\mathscr{P}\left[\bar{N}_i^r, \bar{U}_i^r\right]$, we can compute $\mathscr{P}\left[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r\right]$. Moreover, following from Corollary 2, every configuration $\mathscr{P}\left[\bar{N}_1^r, \bar{U}_1^r\right]$ converges in a finite

number of steps i^* to satisfy

$$\begin{aligned} \mathscr{P}[\bar{N}_i^r, \bar{U}_i^r] = \mathscr{P}[\bar{N}_{i+1}^r, \bar{U}_{i+1}^r] \qquad & \forall i \geq i^* \\ \mathscr{P}[U_i^r > 0] = 0 \qquad & \forall i \geq i^* \end{aligned}$$

It is easy to see that the first step of each round (where the player selects S healthy node and infects them) also defines a Markov chain that deterministically increases, if possible, the infection size by S.

Specifically, the transition probabilities from step 0 to step 1 in each round are defined by:

$$\mathscr{P}[\bar{N}_1^r, \bar{U}_1^r] = \begin{cases} \mathscr{P}[N_0^r = \bar{N}_1^r - S] & \text{iff } \bar{N}_1^r \geq S, \bar{U}_1^r = S \\ \mathscr{P}[N_0^r = \bar{N}_1^r] & \text{iff } \bar{N}_1^r > (N-S), \bar{U}_1^r = 0 \\ 0 & \text{otherwise} \end{cases} \tag{4.18}$$

The distribution underlying the final infection size can be computed as follows:

- The distribution underlying the first step of the game is known:

$$\mathscr{P}[\bar{N}_0^0, \bar{U}_0^0] = \delta_{\bar{N}_0^0,0}\delta_{\bar{U}_0^0,0}$$

- For K rounds:
  - If $r > 0$, then $\mathscr{P}[\bar{N}_0^r, \bar{U}_0^r] = \mathscr{P}[\bar{N}_\infty^{r-1}, \bar{U}_\infty^{r-1}]$.
  - Apply Equation (4.18) to compute $\mathscr{P}[\bar{N}_1^r, \bar{U}_1^r]$.
  - Until convergence:
    - Apply Equation (4.17) to compute $\mathscr{P}[\bar{N}_i^r, \bar{U}_i^r]$.

Part II
Oracular Byzantine Reliable Broadcast

5 Overview

5.1 Introduction

Byzantine reliable broadcast (BRB) is one of the most fundamental and versatile building blocks in distributed computing, powering a variety of Byzantine fault-tolerant (BFT) systems [37, 61]. The BRB abstraction has recently been shown to be strong enough to process payments, enabling cryptocurrency deployments in an asynchronous environment [80]. Originally introduced by Bracha [31] to allow a set of processes to agree on a single message from a designated sender, BRB naturally generalizes to the multi-shot case, enabling higher-level abstractions such as Byzantine FIFO [34, 132] and causal [11, 24] broadcast. We study a practical, multi-shot variant of BRB whose interface is split between broadcasting *clients* and delivering *servers*. We call this abstraction Client-Server Byzantine Reliable Broadcast (CSB).

CSB in brief. Clients broadcast, and servers deliver, *payloads* composed by a *context* and a *message*. This interface allows, for example, Alice to announce her wedding as well as will her fortune by respectively broadcasting

$$\Big(\underbrace{\texttt{"My wife is"}}_{\text{context } c_w}, \underbrace{\texttt{"Carla"}}_{\text{message } m_w}\Big) \quad \Big(\underbrace{\texttt{"All my riches go to"}}_{\text{context } c_r}, \underbrace{\texttt{"Bob"}}_{\text{message } m_r}\Big)$$

CSB guarantees that: (*Consistency*) no two correct servers deliver different messages for the same client and context; (*Totality*) either all correct servers deliver a message for a given client and context, or no correct server does; (*Integrity*) if a correct server delivers a payload from a correct client, then the client has broadcast that payload; and (*Validity*) a payload broadcast by a correct client is delivered by at least one correct server. Following from the above example, Carla being Alice's wife does not conflict with Bob being her sole heir (indeed, $c_w \neq c_r$), but Alice would not be able to convince two correct servers that she married Carla and Diana, respectively. Higher-level broadcast abstractions can be easily built on top of CSB. For example, using integer sequence numbers as contexts and adding a reordering layer yields

Client-Server Byzantine FIFO Broadcast. For the sake of CSB, however, it is not important for contexts to be integers, or satisfy any property other than comparability. Throughout the remainder of this Part, the reader can picture contexts as opaque binary blobs. Lastly, while the set of servers is known, CSB as presented does not assume any client to be known a priori. The set of clients can be permissionless, with servers discovering new clients throughout the execution.

A utopian model. Real-world BRB implementations are often bottlenecked either by expensive signature verifications [52] or by communication overhead [32, 110, 111]. With the goal of broadening those bottlenecks, simplified, more trustful models are useful to establish a (sometimes grossly unreachable) bound on the efficiency that an algorithm can attain in the Byzantine setting. For example, in a utopian model where any agreed-upon process can be trusted to never fail (let us call it an *oracle*), CSB can easily be implemented with great efficiency. Upon initialization, the oracle organizes all clients in a list, which it disseminates to all servers. For simplicity, let us call *id* a client's position in the list. To broadcast a payload p, a client with id i simply sends p to the oracle: the oracle checks p for equivocation (thus ensuring consistency), then forwards (i, p) to all servers (thus ensuring validity and totality). Upon receiving (i, p), a server blindly trusts the oracle to uphold all CSB properties, and delivers (i, p). Oracle-CSB is clearly very efficient. On the one hand, because the oracle can be trusted not to attribute spurious payloads to correct clients, integrity can be guaranteed without any server-side signature verification. On the other, in order to deliver (i, p), a server needs to receive just $(\lceil \log_2(c) \rceil + |p|)$ bits, where c denotes the total number of clients, and $|p|$ measures p's length in bits. This is optimal assuming the rate at which clients broadcast is unknown[1] or uniform[2] [51].

Matching the oracle. Due to its reliance on a single infallible process, Oracle-CSB is not a fault-tolerant distributed algorithm: shifting back to the Byzantine setting, a single failure would be sufficient to compromise all CSB properties. Common sense suggests that Byzantine resilience will necessarily come at some cost: protocol messages must be exchanged to preserve consistency and totality, signatures must be produced and verified to uphold integrity and, lacking the totally-ordering power that only consensus can provide, ids cannot be assigned in an optimally dense way. However, this Part proves the counter-intuitive result that an asynchronous, optimally-resilient, Byzantine implementation of CSB can asymptotically match the efficiency of Oracle-CSB. This is not just up to a constant, but identically. In a synchronous execution, free from Byzantine misbehaviour, and as the number of concurrently

[1] Lacking an assumption on broadcasting rates, an adversarial scheduler could have all messages broadcast by the client with the longest id, which we cannot guarantee to be shorter than $\lceil \log_2(c) \rceil$ bits.

[2] Should some clients be expected to broadcast more frequently than others, we could further optimize Oracle-CSB by assigning smaller ids to more active clients, possibly at the cost of having less active clients have ids whose length exceeds $\lceil \log_2(c) \rceil$. Doing so, however, is beyond the scope of this Part.

broadcasting clients goes to infinity (we call these conditions the *batching limit*[3]), our CSB implementation Draft delivers a payload p at an asymptotic[4], amortized cost of 0 signature verifications[5] and $(\lceil \log_2(c) \rceil + |p|)$ bits exchanged per server, the same as in Oracle-CSB (we say that Draft achieves *oracular efficiency*). At the batching limit a Draft server is dispensed from nearly all signature verifications, as well as nearly all traffic that would be normally required to convey protocol messages, signatures, or client public keys. Network is the limit: payloads are delivered as quickly as they can be received.

CSB's common bottlenecks. To achieve oracular efficiency, we focus on three types of server overhead that commonly affect a real-world implementation of CSB:

- *Protocol overhead.* Safekeeping consistency and totality typically requires some form of communication among servers. This communication can be direct (as in Bracha's original, all-to-all BRB implementation) or happen through an intermediary (as in Bracha's signed, one-to-all-to-one BRB variant), usually employing signatures to establish authenticated, intra-server communication channels through a (potentially Byzantine) relay.
- *Signature overhead.* Upholding integrity usually requires clients to authenticate their messages using signatures. For servers, this entails both a computation and a communication overhead. On the one hand, even using well-optimized schemes, signature verification is often CPU-heavy enough to dominate a server's computational budget, dwarfing in particular the CPU footprint of much lighter, symmetric cryptographic primitives such as hashes and ciphers. On the other hand, transmitting signatures results in a fixed communication overhead per payload delivered. While the size of a signature usually ranges from a few tens to a few hundreds of bytes, this overhead is non-negligible in a context where many clients broadcast small messages. This is especially true in the case of payments, where a message reduces to the identifier of a target account and an integer to denote the amount of money to transfer.
- *Identifier overhead.* CSB's multi-shot nature calls for a sender identifier to be attached to each broadcast payload. Classically, the client's public key is used as identifier. This is convenient for two reasons. First, knowing a client's identifier is sufficient to authenticate its payloads. Second, asymmetric keypairs have very low probability of collision. As such, clients can create identities in the system without any need for coordination: locally generating a keypair is sufficient to begin broadcasting messages. By cryptographic design, however, public keys are sparse, and their size does not change with the

[3]The batching limit includes other easily achievable, more technical conditions that we omit in this section for the sake of brevity. We formally define the batching limit in Section 6.4.2

[4]The asymptotic costs are reached quite fast, at rates comparable to C^{-1} or $log(C) \cdot C^{-1}$.

[5]This does not mean that batches are processed in constant time: hashes and signature aggregations, for example, still scale linearly in the size of a batch. The real-world computational cost of such simple operations, however, is several orders of magnitude lower than that of signature verification.

number of clients. This translates to tens to hundreds of bytes being invested to identify a client from a set that can realistically be enumerated by a few tens of bits. Again, this communication overhead is heavier on systems where broadcasts are frequent and brief.

On the way to matching Oracle-CSB's performance, we develop techniques to negate all three types of overhead: at the batching limit, a Draft server delivers a payload wasting 0 bits to protocol overhead, performing 0 signature verifications, and exchanging $\lceil \log_2(c) \rceil$ bits of identifier, the minimum required to enumerate the set of clients. We outline our contributions below, organized in three (plus one) take-home messages (T-HMs).

T-HM1: The effectiveness of batching goes beyond total order. In the totally ordered setting, batching is famously effective at amortizing protocol overhead [8, 133]. Instead of disseminating its message to all servers, a client hands it over to (one or more)[6] batching processes. Upon collecting a large enough set of messages, a batching process organizes all messages in a batch, which it then disseminates to the servers. Having done so, the batching process submits the batch's hash to the system's totally-ordering primitive. Because hashes are constant in length, the cost of totally ordering a batch does not depend on its size. Once batches are totally ordered, so too are messages (messages within a batch can be ordered by any deterministic function), and equivocations can be handled at the application layer (for example, in the context of a cryptocurrency, the second request to transfer the same asset can be ignored by all correct servers, with no need for additional coordination). At the limit of infinitely large batches, the relative overhead of the ordering protocol becomes vanishingly small, and a server can allocate virtually all of its bandwidth to receiving batches. This strategy, however, does not naturally generalize to CSB, where batches lack total order. As payloads from multiple clients are bundled in the same batch, a correct server might detect equivocation for only a subset of the payloads in the batch. Entirely accepting or entirely rejecting a partially equivocated batch is not an option. In the first case, consistency could be violated. In the second case, a single Byzantine client could single-handedly "poison" the batches assembled by every correct batching process with equivocated payloads, thus violating validity. In Draft, a server can partially reject a batch, acknowledging all but some of its payloads. Along with its partial acknowledgement, a server provides a proof of equivocation to justify each exception. Having collected a quorum of appropriately justified partial acknowledgements, a batching process has servers deliver only those payloads that were not excepted by any server. Because proofs of equivocations cannot be forged for correct clients, a correct client handing over its payload to a correct batching process is guaranteed to have that payload delivered. In the common case where batches have little to no equivocations, servers exchange either empty or small lists of exceptions, whose size does not scale with that of the batch. This extends the protocol-amortizing power of batching to CSB and, we conjecture, other non-totally ordered abstractions.

[6] In most real-world implementations, a client optimistically entrusts its payload to a single process, extending its request to larger portions of the system upon expiration of a suitable timeout.

T-HM2: Interactive multi-signing can slash signature overhead. Traditionally, batching protocols are non-interactive on the side of clients. Having offloaded its message to a correct batching process, a correct client does not need to interact further for its message to be delivered: the batching process collects an arbitrary set of independently signed messages and turns to the servers to get each signature verified, and the batch delivered. This approach is versatile (messages are not tied to the batch they belong to) and reliable (a client crashing does not affect a batch's progress) but expensive (the cost of verifying each signature is high and independent of the batch's size). In Draft, batching processes engage in an interactive protocol with clients to replace, in the good case, all individual signatures in a batch with a single, batch-wide *multi-signature*. In brief, multi-signature schemes extend traditional signatures with a mechanism to *aggregate* signatures and public keys: an arbitrarily large set of signatures for the same message[7] can be aggregated into a single, constant-sized signature; similarly, a set of public keys can be aggregated into a single, constant-sized public key. The aggregation of a set of signatures can be verified in constant time against the aggregation of all corresponding public keys. Unlike verification, aggregation is a cheap operation, reducing in some schemes to a single multiplication on a suitable field. Multi-signature schemes open a possibility to turn expensive signature verification into a once-per-batch operation. Intuitively, if each client contributing to a batch could multi-sign the entire batch instead of its individual payload, all multi-signatures could be aggregated, allowing servers to authenticate all payloads at once. However, as clients cannot predict how their payloads will be batched, this must be achieved by means of an interactive protocol. Having collected a set of individually-signed payloads in a batch, a Draft batching process shows to each contributing client that its payload was included in the batch. In response, clients produce their multi-signatures for the batch's hash, which the batching process aggregates. Clients that fail to engage in this interactive protocol (e.g., because they are faulty or slow) do not lose liveness, as their original signature can still be attached to the batch to authenticate their individual payload. In the good case, all clients reply in a timely fashion, and each server has to verify a single multi-signature per batch. At the limit of infinitely large batches, this results in each payload being delivered at an amortized cost of 0 signature verifications. The usefulness of this interactive protocol naturally extends beyond CSB to all multi-shot broadcast abstractions whose properties include integrity.

T-HM3: Dense id assignment can be achieved without consensus. In order to efficiently convey payload senders, Oracle-CSB's oracle organizes all clients in a list, attaching to each client a successive integral identifier. Once the list is disseminated to all servers, the oracle can identify each client by its identifier, sparing servers the cost of receiving larger, more sparse, client-generated public keys. Id-assignment strategies similar to that of Oracle-CSB can be developed, in the distributed setting, building on top of classical algorithms that identify clients by their full public keys (we call such algorithms *id-free*, as opposed to algorithms such as Draft, which are *id-optimized*). In a setting where consensus can be achieved, the identifier

[7] Some multi-signature schemes also allow the aggregation of signatures on heterogeneous messages. In that case, however, aggregation is usually as expensive as signature verification. Given our goal to reduce CPU complexity for servers, this Part entirely disregards heterogeneous aggregation schemes.

density of Oracle-CSB is easily matched. Upon initialization, each client submits its public key to an id-free implementation of Atomic Broadcast. Upon delivery of a public key, every correct process agrees on its position within the common, totally-ordered log. As in Oracle-CSB, each client can then use its position in the list as identifier within some faster, id-optimized broadcast implementation. In a consensus-less setting, achieving a totally-ordered list of public keys is famously impossible [62]. This Part, however, proves the counter-intuitive result that, when batching is used, the density of ids assigned by a consensus-less abstraction can asymptotically match that of those produced by Oracle-CSB or consensus. In Dibs, our consensus-less id-assigning algorithm, a client requests an id from every server. Each server uses an id-free implementation of FIFO Broadcast to order the client's public key within its own log. Having observed its public key appear in at least one log, the client publicly elects the server in charge of that log to be its *assigner*. Having done so, the client obtains an id composed of the assigner's public key and the client's position within the assigner's log. We call the two components of an id *domain* and *index*, respectively. Because the set of servers is known to (and can be enumerated by) all processes, an id's domain can be represented in $\lceil \log_2(n) \rceil$ bits, where n denotes the total number of servers. Because at most c distinct clients can appear in the FIFO log of any server, indices are at most $\lceil \log_2(c) \rceil$ bits long. In summary, Dibs assigns ids to clients without consensus, at an additional cost of $\lceil \log_2(n) \rceil$ bits per id. Interestingly, even this additional complexity can be amortized by batching. Having assembled a batch, a Draft batching process represents senders not as a list of ids, but as a map, associating to each of the n domains the indices of all ids in the batch under that domain. At the limit of infinitely large batches ($C \gg N$), the bits required to represent the map's keys are entirely amortized by those required to represent its values. This means that, while $\left(\lceil \log_2(n) \rceil + \lceil \log_2(c) \rceil\right)$ bits are required to identify a client in isolation, $\lceil \log_2(c) \rceil$ bits are sufficient if the client is batched: even without consensus, Draft asymptotically matches the id efficiency of Oracle-CSB.

Bonus T-HM: Untrusted processes can carry the system. In THM1, we outlined how batching can be generalized to the consensus-less case, and discussed its role in removing protocol overhead. In THM2, we sketched how an interactive protocol between clients and batching processes can eliminate signature overhead. In employing these techniques, we shifted most of the communication and computation complexity of our algorithms from servers to batching processes. Batching processes verify all client signatures, create batches, verify and aggregate all client multi-signatures, then communicate with servers in an expensive one-to-all pattern, engaging server resources (at the batching limit) as little as an oracle would. Our last contribution is to observe that a batching process plays no role in upholding CSB's safety. As we discuss in detail throughout the remainder of this Part, a malicious batching process cannot compromise consistency (it would need to collect two conflicting quorums of acknowledgements), totality (any server delivering a batch has enough information to convince all others to do the same) or integrity (batches are still signed, and forged or improperly aggregated multi-signatures are guaranteed to be detected). Intuitively, the only damage a batching pro-

cess can do to the system is to refuse to process client payloads[8]. This means that a batching process does not need to satisfy the same security properties as a server. CSB's properties cannot be upheld if a third of the servers are faulty. Conversely, Draft has both liveness and safety as long as *a single* batching process is correct. This observation has profound practical implications. In the real world, scaling the resources of a permissioned, security-critical set of servers can be hard. On the one hand, reputable, dependable institutions partaking in the system might not have the resources to keep up with its demands. On the other, more trusted hardware translates to a larger security cross-section. Trustless processes, however, are plentiful to the point that permissionless cryptocurrencies traditionally waste their resources, making them compete against each other in expensive proofs of Sybil-resistance [122]. In this Part, we extend the classical client-server model with *brokers*, a permissionless, scalable set of processes whose only purpose is to alleviate server complexity. Unlike servers, more than two-thirds of which we assume to be correct, all brokers but one can be faulty. In Draft, brokers act as an intermediary between clients and servers, taking upon themselves the batching of payloads, verification and aggregation of signatures, the dissemination of batches, and the transmission of protocol messages.

Roadmap. We discuss related work in Section 5.2. We state our model and recall useful cryptographic background in Section 5.3. We discuss our CSB implementation Draft in Chapter 6: we formally define the Client-Server Byzantine Reliable Broadcast abstraction in Section 6.1; we present Draft in Section 6.2; we prove Draft's correctness in Section 6.3; we analyze Draft's complexity at the batching limit in Section 6.4. We discuss our Directory implementation Dibs in Chapter 7: we introduce the Directory abstraction in Section 7.1; we present Dibs's pseudocode in Section 7.2; we prove the correctness of Dibs in Section 7.3.

5.2 Related Work

Byzantine Reliable Broadcast (BRB) is a classical primitive of distributed computing, with widespread practical applications such as in State Machine Replication (SMR) [33, 37, 118], Byzantine agreement [44, 92, 100, 123, 141], blockchains [8, 53, 55], and online payments [50, 80, 101]. In classical BRB, a system of n processes agree on a single message from a single *source* (one of the n processes), while tolerating up to f Byzantine failures (f of the n processes can behave arbitrarily). A well known solution to asynchronous BRB with provably optimal resilience ($f < n/3$) was first proposed by Bracha [30, 31] who introduced the problem. Bracha's broadcast reaches $O(n^2)$ message complexity, and $O(n^2 L)$ communication complexity (total number of transmitted bits between correct processes [154]), where L is the length of the message. Since $O(n^2)$ message complexity is provably optimal [59], the main focus of BRB-related research has been on reducing its communication complexity. The best lower bound

[8] Or cause servers to waste resources, e.g., by transmitting improperly signed batches. Simple accountability measures, we conjecture, would be sufficient to mitigate these attacks in Draft. A full discussion of Denial of Service, however, is beyond the scope of this Part.

for communication complexity is $\Omega(nL+n^2)$, although it is unknown whether it is tight. The nL term comes from all processes having to receive the message (length L), while the n^2 term comes from each of the n processes having to receive $\Omega(n)$ protocol messages to ensure agreement in the presence of $f=\Theta(n)$ failures [59]. One line of research focuses on worst-case complexity, predominantly using error correcting codes [17, 130] or erasure codes [5, 38, 85, 126], and has produced various BRB protocols with improved complexity [5, 35, 38, 57, 123], many of them quite recently. The work of Das, Xiang and Ren [57] achieves $O(nL+kn^2)$ communication complexity (specifically, $7nL+2kn^2$), where k is the security parameter (e.g., the length of a hash, typically 256 bits). As the authors note, the value of hidden constants (and k, which is sometimes considered as a constant in literature) is particularly important when considering practical implementations of these protocols. Another line of research focuses on optimizing the good case performance of BRB, i.e., when the network behaves synchronously and no process misbehaves [1, 35, 44, 100, 129]. As the good case is usually the common case, in practice, the real-world communication complexity of these optimistic protocols matches that of the good case. A simple and widely-used hash-based BRB protocol is given by Cachin *et al.* [35]. It replaces the echo and ready phase messages in Bracha's protocol with hashes, achieving $O(nL+kn^2)$ in the good case (specifically, $nL+2kn^2$), and $O(n^2L)$ in the worst-case. Considering practical throughput, some protocols also focus on the *amortized* complexity per source message [44, 111, 129]. Combining techniques such as *batching* [44] and threshold signatures [137], at the limit (of batch size), BRB protocols reach $O(nL)$ amortized communication complexity in the good case [129]. At this point, the remaining problem lies in the hidden constants. In the authenticated setting, batching-based protocols rely on digital signatures to validate (source) messages before agreeing to deliver them [129]. In reality, each source message in a batch includes its content, an identifier of the source (e.g., a k-sized public key), a sequence id (identifying the message), and a k-sized signature. When considering systems where L is small (e.g., online payments), these can take up a large fraction of the communication. To be precise, the good-case amortized communication complexity would be $O(nL+kn)$. In fact, message signatures (the kn factor) are by far the main bottleneck in practical applications of BRB today [55, 141], both in terms of communication and computation (signature verification), leading to various attempts at reducing or amortizing their cost [53, 111]. For example, Crain *et al.* [53] propose *verification sharding*, in which only $f+1$ processes have to receive and verify all message signatures in the good case, which is a 3-fold improvement over previous systems (on the kn factor) where all n processes verify all signatures. However, by itself, this does not improve on the amortized cost of $O(nL+kn)$ per message. When contrasting theoretical research with practical systems, it is interesting to note the gap that can surge between the theoretical model and reality. The recent work of Abraham *et al.* [1], focused on the good-case latency of Byzantine broadcast, expands on some of these mismatches and argues about the practical limitations of focusing on the worst-case. Another apparent mismatch lies in the classical model of Byzantine broadcast. In many of the applications of BRB mentioned previously (e.g., SMR, permissioned blockchains, online payments), there is usually a set of *servers* (n, up to f of which are faulty), and a set of external clients (X) which are the true sources of messages. The usual transformation

from BRB's classical model into these practical settings maps the set of n servers as the n processes and simply excludes clients as system entities, e.g., assuming their messages are relayed through one of the servers. Since the number of clients can be very large ($|X| \gg n$), clients are untrusted (which can limit their usefulness), and the focus is on the communication complexity of the *servers*, this transformation seems reasonable and simplifies the problem. However, it can also limit the search for more practical solutions. In this Part, in contrast with the classical model of BRB, we explicitly include the set of clients X in our system while focusing on the communication complexity surrounding the servers (i.e., the bottleneck). Furthermore, we introduce *brokers*, an untrusted set B of processes, only one of which is assumed to be correct, whose goal is to assist servers in their operation. By doing this, we can leverage brokers to achieve a good-case, amortized communication complexity (for servers, information received or sent) of $nL + o(nL)$.

5.3 Model & background

5.3.1 Model

System and adversary. We assume an asynchronous message-passing system where the set Π of processes is the distinct union of three sets: **servers** (Σ), **brokers** (B), and **clients** (X). We use $n = |\Sigma|$, $k = |B|$ and $c = |X|$. Any two processes can communicate via reliable, FIFO, point-to-point links (messages are delivered in the order they are sent). Faulty processes are Byzantine, i.e., they may fail arbitrarily. Byzantine processes know each other, and may collude and coordinate their actions. At most f servers are Byzantine, with $n = 3f + 1$. At least one broker is correct. All clients may be faulty. We use Π_C and Π_F to respectively identify the set of correct and faulty processes. The adversary cannot subvert cryptographic primitives (e.g., forge signatures). Servers and brokers[9] are permissioned (every process knows Σ and B), clients are permissionless (no correct process knows X a priori). We call *certificate* a statement signed by either a plurality ($f + 1$) or a quorum ($2f + 1$) of servers. Since every process knows Σ, any process can verify a certificate.

Good case. The algorithms presented in this Part are designed to uphold all their properties in the model above. Draft, however, achieves oracular efficiency only in the *good case*. In the good case, links are synchronous (messages are delivered at most one time unit after they are sent), all processes are correct, and the set of brokers contains only one element. To take advantage of the good case, Draft makes use of timers (which is uncommon for purely asynchronous algorithms). A timer with timeout δ set at time t rings: after time $(t + \delta)$, if the system is synchronous; after time t, otherwise. Intuitively, in the non-synchronous case, timers disregard their timeout entirely, and are guaranteed to ring only eventually.

[9]The assumption that brokers are permissioned is made for simplicity, and can be easily relaxed to the requirement that every correct process knows at least one correct broker.

5.3.2 Background

Besides commonly used hashes and signatures, the algorithms presented in this Part make use of two less often used cryptographic primitives, namely, multi-signatures and Merkle trees. We briefly outline their use below. An in-depth discussion of their inner workings, however is beyond the scope of this Part.

Multi-signatures. Like traditional signatures, multi-signatures [15] are used to publicly authenticate messages: a public / secret keypair (p, r) is generated locally; r is used to produce a signature s for a message m; s is publicly verified against p and m. Unlike traditional signatures, however, multi-signatures for the same message can be *aggregated*. Let $(p_1, r_1), \ldots, (p_n, r_n)$ be a set of keypairs, let m be a message, and let s_i be r_i's signature for m. $(p_1, \ldots, p_n)$ and $(s_1, \ldots, s_n)$ can be respectively aggregated into a constant-sized public key $\hat{p}$ and a constant-sized signature $\hat{s}$. As with individually-generated multi-signatures, $\hat{s}$ can be verified in constant time against $\hat{p}$ and m. Aggregation is cheap and non-interactive: provided with $(p_1, \ldots, p_n)$ (resp., $(s_1, \ldots, s_n)$) any process can compute $\hat{p}$ (resp., $\hat{s}$).

Merkle trees. Merkle trees [117] extend traditional hashes with compact *proofs of inclusion*. As with hashes, a sequence $(x_1, \ldots, x_n)$ of values can be hashed into a preimage and collision-resistant digest (or *root*) r. Unlike hashes, however, a proof p_i can be produced from $(x_1, \ldots, x_n)$ to attest that the i-th element of the sequence whose root is r is indeed x_i. In other words, provided with r, p_i and x_i, any process can verify that the i-th element of $(x_1, \ldots, x_n)$ is indeed x_i, without having to learn $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n)$. The size of a proof of inclusion for a sequence of n elements is logarithmic in n.

6 Draft

In this chapter, we present in detail the **Client-Server Byzantine Reliable Broadcast** abstraction and discuss its properties. We then present Draft, an algorithm that implements Client-Server Byzantine Reliable Broadcast, and evaluate its **security** and **complexity**, showing that it achieves oracularity at the batching limit.

6.1 Interface

A **Client-Server Byzantine Reliable Broadcast (CSB) system** offers two interfaces, CSB Client (instance cl) and CSB Server (instance sr), exposing the following events:

- **Request**: $\langle cl.\text{Broadcast} \mid context, message\rangle$: Broadcasts a message $message$ for context $context$ to all servers.
- **Indication** $\langle sr.\text{Deliver} \mid client, context, message\rangle$: Delivers the message $message$ broadcast by client $client$ for context $context$.

A Client-Server Byzantine Reliable Broadcast system satisfies the following properties:

1. **No duplication**: No correct server delivers more than one message for the same client and context.
2. **Integrity**: If a correct server delivers a message m for context c from a correct client χ, then χ previously broadcast m for c.
3. **Consistency**: No two correct servers deliver different messages for the same client and context.
4. **Validity**: If a correct client χ broadcasts a message for context c, then eventually a correct server delivers a message for c from χ.

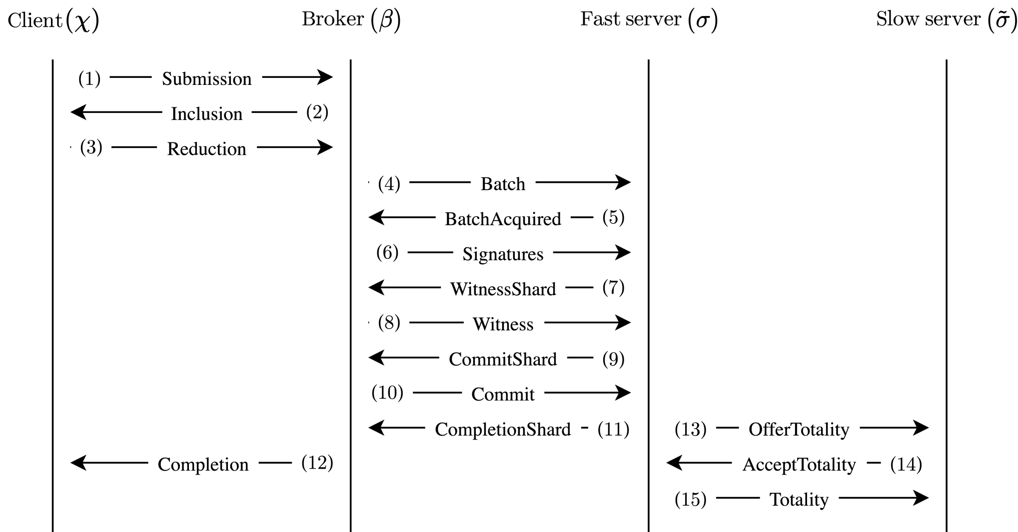


Figure 6.1: Draft's protocol. Having collected a batch of client payloads, a broker engages in an interactive protocol with clients to reduce the batch, replacing (most of) its individual payload signatures with a single, batch-wide multi-signature. The broker then disseminates the batch to all servers, successively gathering a witness for its correctness and a certificate to commit (some of) its payloads. Having had a plurality of servers deliver the batch, the broker notifies all clients with a suitable certificate. In the bad case, servers can ensure totality without any help from the broker, propagating batches and commit certificates in an all-to-all fashion.

5. **Totality**: If a correct server delivers a message for context c from a client χ, then eventually every correct server delivers a message for c from χ.

6.2 Algorithm

In this section, we present our CSB implementation Draft. We outline Draft's protocol in Section 6.2.1, providing qualitative arguments for its security. We qualitatively analyze Draft's complexity in Section 6.2.2. We present Draft's full pseudocode in Sections 6.2.3 (client), 6.2.4 (broker) and 6.2.5 (server). The remainder of Chapter 6 proves Draft's security and complexity to the fullest extent of formal detail.

6.2.1 Protocol & correctness overview

Dramatis personae. The goal of this section is to provide an intuitive understanding of Draft's protocol. In order to do this, we focus on four processes: a correct client χ, a correct broker β, a correct and fast server σ, and a correct but slow server $\tilde{\sigma}$. We follow the messages exchanged between χ, β, σ and $\tilde{\sigma}$ as the protocol unfolds, as captured by Figure 6.1.

The setting. χ's goal is to broadcast a payload p. χ has already used Draft's underlying Directory abstraction (DIR) to obtain an id i. In brief, DIR guarantees that i is assigned to χ only, and provides χ with an *assignment certificate* a, which χ can use to prove that its id is indeed i. As we discussed in Section 5.1, Draft uses DIR-assigned ids to identify payload senders. This is essential to Draft's performance, as DIR guarantees *density*: as we outline in Section 6.2.2, $\lceil \log_2(c) \rceil$ bits are asymptotically sufficient to represent each id in an infinitely large batch. We discuss the details of the DIR abstraction in Section 7, along with our DIR implementation, Dibs. Throughout the remainder of this Part, we say that a process π *knows* an id $\hat{i}$ iff π knows the public keys to which $\hat{i}$ is assigned.

Building a batch. In order to broadcast its payload p, χ produces a signature s for p, and then sends a `Submission` message to β (fig. 6.1, step 1). The `Submission` message contains p, s, and χ's assignment certificate a. Upon receiving the `Submission` message, β learns χ's id i from a, then verifies s against p. Having done so, β stores (i, p, s) in its *submission pool*. For a configurable amount of time, β fills its pool with submissions from other clients, before *flushing* it into a *batch*. Let us use $(i_1, p_1, s_1), \ldots, (i_b, p_b, s_b)$ to enumerate the elements β flushes from the submission pool (for some n, we clearly have $(i, p, s) = (i_n, p_n, s_n)$). For convenience, we will also use χ_j to identify the sender of p_j (owner of i_j). Importantly, β flushes the pool in such a way that $i_j \neq i_k$ for all $j \neq k$: for safety reasons that will soon be clear, Draft's protocol prevents a client from having more than one payload in any specific batch. Because of this constraint, some payloads might linger in β's pool. This is not an issue: β will simply flush those payloads to a different batch at a later time. When building the batch, β splits submissions and signatures, storing $(i_1, p_1), \ldots, (i_b, p_b)$ separately from $s_1, \ldots, s_b$.

Reducing the batch. Having flushed submissions $(i_1, p_1), \ldots, (i_b, p_b)$ and signatures $s_1, \ldots, s_b$, β moves on to *reduce* the batch, as exemplified in Figure 6.2. In an attempt to minimize signature overhead for servers, β engages in an interactive protocol with clients $\chi_1, \ldots, \chi_b$ to replace as many signatures as possible with a single, batch-wide multi-signature. In order to do so, β organizes $(i_1, p_1), \ldots, (i_b, p_b)$ in a Merkle tree with root r (for brevity, we call r the batch's root). β then sends an `Inclusion` message to each χ_j (fig. 6.1, step 2). Each `Inclusion` message contains r, along with a proof of inclusion q_j for (i_j, p_j). Upon receiving its `Inclusion` message, χ checks q_n against r. In doing so, χ comes to two conclusions. First, χ's submission $(i, p) = (i_n, p_n)$ is part of a batch whose root is r. Second, because no Draft batch can contain multiple payloads from the same client, that batch does not attribute χ any payload other than p. In other words, χ can be certain that β will not broadcast some spurious payload $p' \neq p$ in χ's name: should β attempt to do that, the batch would be verifiably malformed, and immediately discarded. This means χ can safely produce a multi-signature m for r: as far as χ is concerned, the batch with root r upholds integrity. Having signed r, χ sends m to β by means of a `Reduction` message (fig. 6.1, step 3). Upon receiving χ_j's `Reduction` message, β checks χ_j's multi-signature m_j against r. Having done so, β discards χ_j's original signature s_j. Intuitively, with m_j, χ_j attested its agreement with whatever payload the batch

attributes to χ_j. Because this is equivalent to individually authenticating p_j, s_j is redundant and can be dropped. Upon expiration of a suitable timeout, β stops collecting `Reduction` messages: clearly, if β waited for every χ_j to produce m_j, a single Byzantine client could prevent the protocol from moving forward by refusing to send its `Reduction` message. β aggregates all the multi-signatures it collected for r into a single, batch-wide multi-signature m. In the good case, every χ_j is correct and timely. If so, β drops all individual signatures, and the entire batch is authenticated by m alone.

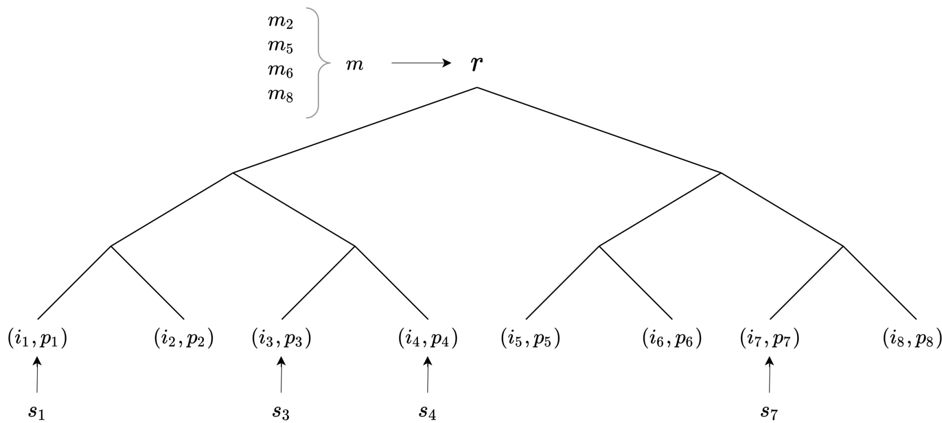


Figure 6.2: An example of partially reduced batch. $B = 8$ submissions are organized on the leaves of a Merkle tree with root r. Each submission (i_j, p_j) is originally authenticated by an individual signature s_j. Upon collecting a multi-signature m_j for r, the broker drops s_j. Here the broker collected multi-signatures m_2, m_5, m_6 and m_8, leaving a *straggler set* $S = \{(i_1, s_1), (i_3, s_3), (i_4, s_4), (i_7, s_7)\}$. Upon expiration of a suitable timeout, the broker aggregates m_2, m_5, m_6 and m_8 into a single multi-signature m. As such, every payload in the batch is authenticated either by m or by S.

The perks of a reduced batch. Having reduced the batch, β is left with a sequence of submissions $(i_1, p_1), \ldots, (i_b, p_b)$, a multisignature m on the Merkle root r of $(i_1, p_1), \ldots, (i_b, p_b)$, and a *straggler* set S holding the individual signatures that β failed to reduce. More precisely, S contains (i_j, s_j) iff β did not receive a valid `Reduction` message from χ_j before the reduction timeout expired. We recall that m's size is constant, and S is empty in the good case. Once reduced, the batch is cheap to authenticate: it is sufficient to verify the batch's multi-signature against the batch's root, and each straggler signature against its individual payload. More precisely, let T denote the set of *timely* clients (χ_j is in T iff $(i_j, ..)$ is not in S). Let t denote the aggregation of T's public keys. Provided with $(i_1, p_1), \ldots, (i_b, p_b)$, m and S, any process that knows $i_1, \ldots, i_b$ can verify that the batch upholds integrity by: (1) computing r and t from $(i_1, p_1), \ldots, (i_b, p_b)$ and S; (2) using t to verify m against r; and (3) verifying each s_j in S against p_j. In the good case, authenticating the batch reduces to verifying a single multi-signature. This is regardless of the batch's size.

The pitfalls of a reduced batch. As we discussed in the previous paragraph, reducing a batch makes it cheaper to verify its integrity. Reduction, however, hides a subtle trade-off: once reduced, a batch gets easier to authenticate *as whole*. Its *individual payloads*, however, become harder to authenticate. For the sake of simplicity, let us imagine that β successfully dropped all the individual signatures it originally gathered from $\chi_1, \ldots, \chi_b$. In order to prove that $(\chi = \chi_n)$ broadcast $(p = p_n)$, β could naively produce the batch's root r, (i_n, p_n)'s proof of inclusion q_n, and the batch's multi-signature m for r. This, however, would not be sufficient to authenticate p: because the multi-signature m_n that χ produced for r was aggregated with all others, m can only be verified by the aggregation of *all* $\chi_1, \ldots, \chi_b$'s public keys. This makes authenticating p as expensive as authenticating the entire batch: in order to verify m, *all* (i_j, p_j) must be produced and checked against r, so that all corresponding public keys can be safely aggregated.

Witnessing the batch. As we discuss next, proving the integrity of individual payloads is fundamental to ensure Draft's validity. In brief, to prove that some χ_k equivocated its payload $p_k = (c_k, l_k)$, a server must prove to β that χ_k also issued some payload $p'_k = (c_k, l'_k \neq l_k)$. Lacking this proof, a single Byzantine server could, for example, claim without basis that χ equivocated p. This could trick β into excluding p, thus compromising Draft's validity. As we discussed in the previous paragraph, however, proving the integrity of an individual payload in a reduced batch is difficult. While we conjecture that purely cryptographic solutions to this impasse might be achievable in some schemes[1], Draft has β engage in a simple protocol to further simplify the batch's authentication, replacing all client-issued (multi-)signatures with a single, server-issued certificate. Having collected and reduced the batch, β sends a `Batch` message to all servers (fig. 6.1, step 4). The `Batch` message only contains $(i_1, p_1), \ldots, (i_b, p_b)$. Upon receiving the `Batch` message, σ collects in a set U_σ all the ids it does not know (i_j is in U_σ iff σ does not know i_j), and sends U_σ back to β by means of a `BatchAcquired` message (fig. 6.1, step 5). Upon receiving σ's `BatchAcquired` message, β builds a set A_σ containing all id assignments that σ is missing (a_j is in A_σ iff i_j is in U_σ). Having done so, β sends a `Signatures` message to σ (fig. 6.1, step 6). The `Signatures` message contains the batch's multi-signature m, the straggler set S, and A_σ. We underline the importance of sending id assignments upon request only. Thinking to shave one round-trip off the protocol, β could naively package in a single message all submissions, all (multi-)signatures, and all assignments relevant to the batch. In doing so, however, β would force each server to receive one assignment per submission, immediately forfeiting Draft's oracular efficiency. As we discuss in Section 6.4.2, at the batching limit we assume that all servers already know all broadcasting clients. In that case, both U_σ and A_σ are constant-sized, empty sets, adding

[1] For example, using BLS, β could aggregate the public keys of $\chi_1, \ldots, \chi_{n-1}, \chi_{n+1}, \ldots, \chi_b$ into a public key $\tilde{t}_n$, then show that the aggregation of $\tilde{t}_n$ with χ's public key correctly verifies m against r. Doing so, however, would additionally require β to exhibit a proof that $\tilde{t}_n$ is not a rogue public key, i.e., that $\tilde{t}_n$ indeed results from the aggregation of client public keys. This could be achieved by additionally having $\chi_1, \ldots, \chi_b$ multi-sign some hard-coded statement to prove that they are not rogues. β could aggregate such signatures on the fly, producing a rogue-resistance proof for $\tilde{t}_n$ that can be transmitted and verified in constant time. This, however, is expensive (and, frankly, at the limit of our cryptographic expertise).

only a vanishing amount of communication complexity to the protocol. Upon receiving the `Signatures` message, σ verifies and learns all assignments in A_σ. Having done so, σ knows $i_1, \ldots, i_b$. As we outlined above, σ can now efficiently authenticate the whole batch, verifying m against the batch's root r, and each i_j in S against p_j. Having established the integrity of the whole batch, σ produces a *witness shard* for the batch, i.e., a multi-signature w_σ for [`Witness`, r], effectively affirming to have successfully authenticated the batch. σ sends w_σ back to β by means of a `WitnessShard` message (fig. 6.1, step 7). Having received a valid `WitnessShard` message from $f+1$ servers, β aggregates all witness shards into a *witness* w. Because w is a plurality $(f+1)$ certificate, at least one correct server necessarily produced a witness shard for the batch. This means that at least one correct server has successfully authenticated the batch by means of client (multi-)signatures. Because w could not have been gathered if the batch was not properly authenticated, w itself is sufficient to authenticate the batch, and β can drop all (now redundant) client-generated (multi-)signatures for the batch. Unlike m, w is easy to verify, as it is signed by only $f+1$, globally known servers. Like m, w authenticates r. As such, any p_j can now be authenticated just by producing w, and (i_j, p_j)'s proof of inclusion q_j.

Gathering a commit certificate. Having successfully gathered a witness w for the batch, β sends w to all servers by means of a `Witness` message (fig. 6.1, step 8). Upon receiving the `Witness` message, σ moves on to check $(i_1, p_1), \ldots, (i_b, p_b)$ for equivocations. More precisely, σ builds a set of *exceptions* E_σ containing the ids of all equivocating submissions in the batch (i_j is in E_σ iff σ previously observed χ_j submit a payload p'_j that conflicts with p_j; we recall that p_j and p'_j conflict if their contexts are the same, but their messages are different). σ then produces a *commit shard* for the batch, i.e., a multi-signature c_σ for [`Commit`, r, E_σ], effectively affirming that σ has found all submissions in the batch to be non-equivocated, *except* for those in E_σ. In the good case, every client is correct and E_σ is empty. Having produced c_σ, σ moves on to build a set Q_σ containing a *proof of equivocation* for every element in E_σ. Let us assume that σ previously received from some χ_k a payload p'_k that conflicts with p_k. σ must have received p'_k as part of some witnessed batch. Let r'_k identify the root of p'_k's batch, let w'_k identify r'_k's witness, let q'_k be (i_k, p'_k)'s proof of inclusion in r'_k. By exhibiting (r'_k, w'_k, p'_k), σ can prove to β that χ_k equivocated: p_k conflicts with p'_k, and (i_k, p'_k) is provably part of a batch whose integrity was witnessed by at least one correct server. Furthermore, because correct clients never equivocate, (r'_k, w'_k, p'_k) is sufficient to convince β that χ_k is Byzantine. For each i_j in E_σ, σ collects in Q_σ a proof of equivocation $\left(r'_j, w'_j, p'_j\right)$. Finally, σ sends a `CommitShard` message back to β (fig. 6.1, step 9). The `CommitShard` message contains c_σ, E_σ and Q_σ. Upon receiving σ's `CommitShard` message, σ verifies c_σ against r and E_σ, then checks all proofs in Q_σ. Having collected valid `CommitShard` messages from a quorum of servers $\sigma_1, \ldots, \sigma_{2f+1}$, β aggregates all commit shards into a *commit certificate* c. We underline that each σ_j signed the same root r, but a potentially different set of exceptions E_{σ_j}. Let E denote the union of $E_{\sigma_1}, \ldots, E_{\sigma_{2f+1}}$. We call E the batch's *exclusion* set. Because a proof of equivocation cannot be produced against a correct client, β knows that all clients identified by E are necessarily

Byzantine. In particular, because χ is correct, $(i = i_n)$ is guaranteed to not be in E.

Committing the batch. Having collected a commit certificate c for the batch, β sends c to all servers by means of a `Commit` message (fig. 6.1, step 10). Upon receiving the `Commit` message, σ verifies c, computes the exclusion set E, then delivers every payload p_j whose id i_j is not in E. Recalling that c is assembled from a quorum of commit shards, at least $f+1$ correct servers contributed to c. This means that, if some i_k is not in E, then at least $f+1$ correct servers found p_k not to be equivocated. As in most BRB implementations [31], this guarantees that no two commit certificates can be gathered for equivocating payloads: Draft's consistency is upheld.

The role of equivocation proofs. As the reader might have noticed, β does not attach any proof of equivocation to its `Commit` message. Having received β's commit certificate c, σ trusts β's exclusion set E, ignoring every payload whose id is in E. This is not because σ can trust β to uphold validity. On the contrary, σ has *no way* to determine that β is not maliciously excluding the payload of a correct client. Indeed, even if σ were to verify a proof of exclusion for every element in E, a malicious β could still censor a correct client simply by ignoring its `Submit` message in the first place. Equivocation proofs are fundamental to Draft's validity not because they *force* malicious brokers to uphold validity, but because they *enable* correct brokers to do the same. Thanks to equivocation proofs, a malicious server cannot trick a correct broker into excluding the payload of a correct client. This is enough to guarantee validity. As we discuss below, χ successively submits p to all brokers until it receives a certificate attesting that p was delivered by at least one correct server. Because we assume at least one broker to be correct, χ is eventually guaranteed to succeed.

Notifying the clients. Having delivered every payload whose id is not in the exclusion set E, σ produces a *completion shard* for the batch, i.e., a multi-signature z_σ for $[\texttt{Completion}, r, E]$, effectively affirming that σ has delivered all submissions in the batch whose id is not in E. σ sends z_σ to β by means of a `CompletionShard` message (fig. 6.1, step 11). Upon receiving $f+1$ valid `CompletionShard` messages, β assembles all completion shards into a *completion certificate* z. Finally, β sends a `Completion` message to $\chi_1, \ldots, \chi_b$ (fig. 6.1, step 12). The `Completion` message contains z and E. Upon receiving the `Completion` message, χ verifies z against E, then checks that i is not in E. Because at least one correct server contributed a completion shard to z, at least one correct process delivered all payloads that E did not exclude, including p. Having succeeded in broadcasting p, χ does not need to engage further, and can stop successively submitting p to all brokers.

No one is left behind. As we discussed above, upon receiving the commit certificate c, σ delivers every payload in the batch whose id is not in the exclusion set E. Having gotten at least one correct server to deliver the batch, β is free to disengage, and moves on to assembling

and brokering its next batch. In a moment of asynchrony, however, all communications between β and $\tilde{\sigma}$ might be arbitrarily delayed. This means that $\tilde{\sigma}$ has no way of telling whether or not it will eventually receive batch and commit certificate: a malicious β might have deliberately left $\tilde{\sigma}$ out of the protocol. Server-to-server communication is thus required to guarantee totality. Having delivered the batch, σ waits for an interval of time long enough for all correct servers to deliver batch and commit certificate, *should the network be synchronous and β correct.* σ then sends to all servers an `OfferTotality` message (fig. 6.1, step 13). The `OfferTotality` message contains the batch's root r, and the exclusion set E. In the good case, upon receiving σ's `OfferTotality` message, every server has delivered the batch and ignores the offer. This, however, is not the case for slow $\hat{\sigma}$, which replies to σ with an `AcceptTotality` message (fig. 6.1, step 14). Upon receiving $\tilde{\sigma}$'s `AcceptTotality` message, σ sends back to $\tilde{\sigma}$ a `Totality` message (fig. 6.1, step 15). The `Totality` message contains all submissions $(i_1, p_1), \ldots, (i_b, p_b)$, id assignments for $i_1, \ldots, i_b$, and the commit certificate c. Upon delivering σ's `Totality` message, $\tilde{\sigma}$ computes r from $(i_1, p_1), \ldots, (i_b, p_b)$, checks c against r, computes E from c, and delivers every payload p_j whose id i_j is not in E. This guarantees totality and concludes the protocol.

6.2.2 Complexity overview

Directory density. As we introduced in Section 6.2.1, Draft uses ids assigned by its underlying Directory (DIR) abstraction to identify payload senders. A DIR-assigned id is composed of two parts: a *domain* and an *index*. Domains form a finite set $\mathbb{D}$ whose size does not increase with the number of clients, indices are natural numbers. Along with safety (e.g., no two processes have the same id) and liveness (e.g., every correct client that requests an id eventually obtains an id), DIR guarantees *density*: the index part of any id is always smaller than the total number of clients c (i.e., each id index is between 0 and $(c-1)$). Intuitively, this echoes the (stronger) density guarantee provided by Oracle-CSB, the oracle-based implementation of CSB we introduced in Section 5.1 to bound Draft's performance. In Oracle-CSB, the oracle organizes all clients in a list, effectively labeling each client with an integer between 0 and $(c-1)$. In a setting where consensus cannot be achieved, agreeing on a totally-ordered list of clients is famously impossible: a consensus-less DIR implementation cannot assign ids if $|\mathbb{D}| = 1$. As we show in Section 7, however, DIR can be implemented without consensus if servers are used as domains ($\mathbb{D} = \Sigma$). In our DIR implementation Dibs, each server maintains an independent list of public keys. In order to obtain an id, a client χ has each server add its public key to its list, then selects a server σ to be its *assigner*. In doing so, χ obtains an id (σ, n), where $n \in 0..(c-1)$ is χ's position in σ's log. In summary, a consensus-less implementation of DIR still guarantees that indices will be smaller than c, at the cost of a non-trivial domain component for each id. This inflates the size of each individual id by $\lceil \log_2(|\mathbb{D}|) \rceil$ bits.

Batching ids. While DIR-assigned ids come with a non-trivial domain component, the size overhead due to domains vanishes when infinitely many ids are organized into a batch. This

is because domains are constant in the number of clients. Intuitively, as infinitely many ids are batched together, repeated domains become compressible. When building a batch, a Draft broker represents the set I of sender ids not as a list, but as a map $\tilde{i}$. To each domain, $\tilde{i}$ associates all ids in I under that domain (n is in $\tilde{i}[d]$ iff (d, n) is in I). Because $\tilde{i}$'s keys are fixed, as the size of I goes to infinity, the bits required to represent $\tilde{i}$'s keys are completely amortized by those required to represent $\tilde{i}$'s values. At the batching limit, the cost of representing each id in $\tilde{i}$ converges to that of representing its index only, $\lceil \log_2(c) \rceil$.

Protocol cost. As we discuss in Section 6.4.2, at the batching limit we assume a good-case execution: links are synchronous, all processes are correct, and the set of brokers contains only one element. We additionally assume that infinitely many clients broadcast concurrently. Finally, we assume all servers to already know all broadcasting clients. Let β denote the only broker. As all broadcasting clients submit their payloads to β within a suitably narrow time window, β organizes all submissions into a single batch with root r. Because links are synchronous and all clients are correct, every broadcasting client submits its multi-signature for r in time. Having removed all individual signatures from the batch, β is left with a single, aggregated multi-signature m and an empty straggler set S. β compresses the sender ids and disseminates the batch to all servers. As m and S are constant-sized, the amortized cost for a server to receive each payload p is $(\lceil \log_2(c) \rceil + |p|)$ bits. As m authenticates the entire batch, a server authenticates each payload at an amortized cost of 0 signature verifications. The remainder of the protocol unfolds as a sequence of constant-sized messages: because all broadcasting clients are known to all servers, no server requests any id assignment; witnesses are always constant-sized; and because all processes are correct, no client equivocates and all exception sets are empty. Finally, again by the synchrony of links, all offers of totality are ignored. In summary, at the batching limit a server delivers a payload at an amortized cost of 0 signature verifications and $(\lceil \log_2(c) \rceil + |p|)$ bits exchanged.

Latency. As depicted in Figure 6.1, the latency of Draft is 10 message delays in the synchronous case (fast servers deliver upon receiving the broker's Commit message), and at most 13 message delays in the asynchronous case (slow servers deliver upon receiving other servers' Totality messages). By comparison, the latency of the optimistic reliable broadcast algorithm by Cachin *et al.* [35] is respectively 4 message delays (synchronous case) and 6 message delays (asynchronous case). Effectively, Draft trades oracular efficiency for a constant latency overhead.

Worst-case complexity. In the worst case, a Draft server delivers a b-bits payload by exchanging $O\big((\log(c) + b)kn\big)$ bits, where c, k and n respectively denote the number of clients, brokers and servers. In brief, the same id, payload and signature is included by each broker in a different batch (hence the k term) and propagated in an all-to-all fashion (carried by `Totality` messages) across correct servers (hence the n term). By comparison, the worst-case

communication complexity of Cachin *et al.*'s optimistic reliable broadcast is $O(ln)$ per server, where l is the length of the broadcast payload. A direct batched generalization of the same algorithm, however, would raise the worst-case communication to $O(ln^2)$ per server, similar to that of Draft when $n \sim k$. Both batched Bracha and Draft can be optimized by polynomial encoding, reducing their per-server worst-case complexity to $O(ln)$ and $O((\log(c) + b)k)$ respectively. Doing so for Draft, however, is beyond the scope of this Part.

6.2.3 Pseudocode (Client)

```
1  implements:
2      CSBClient, instance cl
3
4
5  uses:
6      Directory, instance dir
7      PerfectPointToPointLinks, instance pl
8
9
10 parameters:
11     b: Interval // Batching window
12
13
14 struct Submission:
15     message: Message,
16     signature: Signature,
17     submitted_to: {Broker},
18     included_in: {Root}
19
20
21 upon <cl.Init>:
22     trigger <dir.Signup>;
23     await <dir.SignupComplete | id>;
24
25     submissions: {Context: Submission} = {};
26     completed: {Root} = {};
27
28
29 upon <cl.Broadcast | context, message>:
30     signature = sign([Message, context, message]);
31
32     submissions[context] = Submission {
           message, signature,
```

```
        submitted_to: {}, included_in: {}
    };
33
34  submit(context):
35
36
37 procedure submit(context):
38     if let submission = submissions[context]
         and let β in (B \ submission.submitted_to):
39         assignment = dir.export(id);
40         trigger <pl.Send | β,
               [Submission, assignment, (context, submission.message,
     submission.signature)]>;
41
42         submission.sumbitted_to.add(β);
43         trigger <timer.Set | [Submit, context], 13 + b>;
44
45
46 upon <timer.Ring | [Submit, context]>:
47     submit(context);
48
49
50 upon <pl.Deliver | β, [Inclusion, context, root, proof]>:
51     if let submission = submissions[context]:
52         if verify(root, proof,(id, context, submission.message)):
53             submission.included_in.add(root);
54             multisignature = multisign([Reduction, root]);
55             trigger <pl.Send | β, [Reduction, root, multisignature]>;


56
57
58 upon <pl.Deliver | β, [Completion, root, exclusions, certificate]>:
59     if verify_plurality(certificate, [Completion, root, exclusions])
         and id not in exclusions:
60         completed.add(root);
61
62
63  upon exists (context, submission) in submissions such that
     (submission.included_in ∩ completed) != {}:
64     submissions.remove(context);
```

6.2.4 Pseudocode (Broker)

```
1  implements:
2      CSBBroker, instance bk
3
4
5  uses:
6      Directory, instance dir
7      PerfectPointToPointLinks, instance pl
8
9
10 parameters:
11     b: Interval // Batching window
12
13
14 struct Submission:
15     context: Context,
16     message: Message,
17     signature: Signature
18
19
20 enum Batch:
21     payloads: {Id: (Context, Message)},
22     signatures: {Id: Signature},
23     reductions: {Id: MultiSignature},
24
25     commit_to: {Server},
26     committable: bool,
27
28     variant Reducing:
29         (empty)
30
31     variant Witnessing:
32         witnesses: {Server: MultiSignature}
33
34     variant Committing:
35         commits: {Server: ({Id}, MultiSignature)}
36
37     variant Completing:
38         exclusions: {Id},
39         completions: {Server: MultiSignature}
40
41
```

```
42 upon <bk.Init>:
43     pending: {Id: [Submission]} (default []) = {};
44     pool: {Id: Submission} = {};
45     collecting: bool = false;
46
47     batches: {Root: Batch} = {};
48
49
50 upon <pl.Deliver | χ, [Submission, assignment, (context, message,
   signature)]>:
51     dir.import(assignment);
52
53     if χ.verify(signature, [Message, context, message]) and let id =
   dir[χ]:
54         pending[id].push_back(Submission {context, message, signature
   });
55
56
57 upon exists id in pending such that pending[id] != [] and id not in
   pool:
58     submission = pending[id].pop_front();
59     pool[id] = submission;
60
61
62 upon pool != {} and collecting = false:
63     collecting = true;
64     trigger <timer.Set | [Flush], b + 1>;
65
66
67 upon <timer.Ring | [Flush]>:
68     collecting = false;
69
70     submissions = pool;
71     pool = {};
72
73     leaves = [(id, context, message )
         for (id, Submission {context, message, ..}) in submissions];
74
75     tree = merkle_tree(leaves);
76     root = tree.root();
77
78     for (id, Submission {context, message, ..}) in submissions:
```

```
79          χ = dir[id];
80          proof = tree.prove((id, context, message));
81          trigger <pl.Send | χ, [Inclusion, context, root, proof]>;
82
83      payloads = {id: (context, message)
          for (id, Submission {context, message, ..}) in submissions};
84
85      signatures = {id: signature for (id, Submission {signature, ..})
      in submissions};
86
87      batches[root] = Reducing {
                          payloads, signatures,
                          reductions: {},
                          commit_to: {},
                          committable: false
                      };
88
89      trigger <timer.Set | [Reduce, root], 2>;
90
91
92  upon <pl.Deliver | χ, [Reduction, root, multisignature]>:
93      if let batch alias batches[root] and batch is Reducing
          and let id = dir[χ] and id in batch.payloads:
94          if χ.multiverify(multisignature, [Reduction, root]):
95              batch.reductions[id] = multisignature;
96              batch.signatures.remove(id);
97
98
99  upon <timer.Ring | [Reduce, root]>:
100     batch alias batches[root];
101     compressed_ids = compress(batch.payloads.keys());
102     payloads = batch.payloads.values()
103
104     for σ in Σ:
105         trigger <pl.Send | [Batch, compressed_ids, payloads]>;
106
107     batch = Witnessing {witnesses: {}, ..batch};
108     trigger <timer.Set | [Committable, root], 4>;
109
110
111 upon event <timer.Ring | [Committable, root]>:
112     if let batch = batches[root]:
```

```
113         batch.committable = true;
114
115
116 upon <pl.Deliver | σ, [BatchAcquired, root, unknowns]>:
117     if let batch = batches[root]:
118         if exists unknown in unknowns such that unknown not in dir:
119             return;
120
121         assignments = dir.export(unknowns..);
122
123         multisignature = aggregate(batch.reductions.values());
124         signatures = batch.signatures;
125
126         trigger <pl.Send | σ,
              [Signatures, root, assignments, multisignature, signatures
    ]>;
127
128
129 upon <pl.Deliver | σ, [WitnessShard, root, shard]>:
130     if let batch alias batches[root]:
131         batch.commit_to.add(σ);
132
133         if batch is Witnessing:
134             if σ.multiverify(shard, [Witness, root]):
135                 batch.witnesses[σ] = shard;
136
137
138 upon exists root in batches such that batches[root] is Witnessing
      and |batches[root].witnesses| >= f + 1:
139     batch alias batches[root];
140     certificate = aggregate(batch.witnesses);
141
142     for σ in Σ:
143         trigger <pl.Send | [Witness, root, certificate]>;
144
145     batch = Committing {commits: {}, ..batch};
146
147
148 upon <pl.Deliver | σ, [CommitShard, root, conflicts, commit]>:
149     if let batch alias batches[root] and batch is Committing:
150         if !σ.multiverify(commit, [Commit, root, conflicts.keys()]):
151             return;
```

```
152
153         for (id, (conflict_root, conflict_witness, conflict_proof,
    conflict_message))
            in conflicts:
154             if id not in batch.payloads:
155                 return;
156
157             (context, message) = batch.payloads[id];
158
159             if not verify_plurality(conflict_witness, [Witness,
    conflict_root]):
160                 return;
161
162             if not verify(conflict_root, conflict_proof,
                  (id, context, conflict_message)):
163                 return;
164
165             if message = conflict_message:
166                 return;
167
168         batch.commits[σ] = (conflicts.keys(), commit);
169
170
171 upon exists root in batches such that batches[root] is Committing
        and batches[root].committable and |batches[root].commits| >= 2f +
    1:
172
173     batch alias batches[root];
174     patches: {{Id}: {MultiSignature}} (default {}) = {};
175
176     for (_, (conflicts, commit)) in batch.commits:
177         patches[conflicts].add(commit);
178
179     patches = {ids: aggregate(signatures) for (ids, signatures) in
    patches};
180
181     for σ in batch.commit_to:
182         trigger <pl.Send | σ, [Commit, root, patches]>;
183
184     exclusions = union(patches.keys());
185     batch = Completing {exclusions, ..batch};
186
```

```
187
188 upon <pl.Deliver | σ, [CompletionShard, root, completion]>:
189     if let batch alias batches[root] and batch in Completing:
190         if multiverify(completion, [Completion, root, batch.exclusions
    ]):
191             batch.completions[σ] = completion
192
193
194 upon exists root in batches such that batches[root] is Completing
    and |batches[root].completions| >= f + 1:
195     batch alias batches[root];
196     certificate = aggregate(batch.completions);
197
198     for id in payloads:
199         χ = dir[id];
200         trigger <pl.Send | χ, [Completion, root, batch.exclusions,
    certificate]>;
201
202     batches.remove(root);
```

6.2.5 Pseudocode (Server)

```
1  implements:
2      CSBServer, instance sr
3
4
5  uses:
6      Directory, instance dir
7      PerfectPointToPointLinks, instance pl
8
9
10 struct Batch:
11     ids: [Id],
12     payloads: [(Context, Message)],
13     tree: MerkleTree
14
15
16 upon event <sr.Init>:
17     batches: {Root: Batch} = {};
18     witnesses: {Root: Certificate} = {};
19     commits: {(Root, {Id}): {{Id}: Certificate}} = {};
20
```

```
21      messages: {(Id, Context): (Message, Root)} = {};
22      delivered: {(Id, Context)} = {};
23
24
25  procedure compress(ids):
26      compressed_ids: {Domain: {Id}} (default {}) = {};
27
28      for (domain, index) in ids:
29          compressed_ids[domain].add(index)
30
31      return compressed_ids
32
33  procedure expand(compressed_ids):
34      ids = {};
35
36      for (domain, indices) in compressed_ids:
37          for index in indices:
38              ids.push_back((domain, index));
39
40      ids.sort();
41      return ids;
42
43
44  procedure join(ids, payloads):
45      return [(id, context, message)
          for (id, (context, message)) in zip(ids, payloads)];
46
47
48  procedure handle_batch(compressed_ids, payloads):
49      ids = expand(compressed_ids);
50
51      if ids.has_duplicates() or |ids| != |payloads|:
52          return ⊥;
53
54      unknowns = {id for id in ids such that id not in dir};
55
56      leaves = join(ids, payloads);
57      tree = merkle_tree(leaves);
58      root = tree.root();
59
60      batches[root] = Batch {ids, payloads, tree};
61      return [BatchAcquired, root, unknowns];
```

```
62
63
64 upon event <pl.Deliver | π, [Batch, compressed_ids, payloads]>:
65     response = handle_batch(compressed_ids, payloads);
66
67     if response != ⊥ and π in B:
68         trigger <pl.Send | π, response>;
69
70
71 procedure handle_signatures(root, assignments, multisignature,
     signatures):
72     dir.import(assignments..);
73
74     if root not in batches:
75         return ⊥;
76
77     Batch {ids, payloads, tree} = batches[root];
78
79     if exists id in ids such that id not in dir:
80         return ⊥;
81
82     for (id, signature) in signatures:
83         if id not in ids:
84             return ⊥;
85
86         (context, message) = payloads[ids.index_of(id)];
87
88         if not dir[id].verify(signature, [Message, context, message]):
89             return ⊥;
90
91     multisigners = ids \ signatures.keys();
92
93     if not dir[multisigners..].multiverify(multisignature, [Reduction,
      tree.root()]):
94         return ⊥;
95
96     shard = multisign([Witness, root]);
97     return [WitnessShard, root, shard];
98
99
100 upon <pl.Deliver | β, [Signatures, root, assignments, multisignature,
```

```
     signatures]>:
101      response = handle_signatures(root, assignments, multisignature,
    signatures);
102
103      if response != ⊥:
104          trigger <pl.Send | β, response>;
105
106
107  procedure handle_witness(root, certificate):
108      if root not in batches:
109          return ⊥;
110
111      Batch {ids, payloads, tree} = batches[root];
112
113      if !verify_plurality(certificate, [Witness, root]):
114          return ⊥;
115
116      witnesses[root] = certificate;
117      conflicts: {Id: (Root, Certificate, MerkleProof, Message)} = {};
118
119      for (id, context, message) in join(ids, payloads):
120          if let (original_message, original_root) = messages[(id,
    context)]
            and original_message != message:
121              original_witness = witnesses[original_root];
122              original_batch = batches[original_root];
123
124              Batch {
                     ids: original_ids,
                     payloads: original_payloads,
                     tree: original_tree
                 } = original_batch;
125
126              original_leaf = (id, context, original_message);
127
128              conflicts[id] = (original_root,
                                  original_witness,
                                  original_tree.prove(original_leaf),
                                  original_message
                                 );
129          else:
130              messages[(id, context)] = (message, root);
```

```
131
132     commit = multisign([Commit, root, conflicts.keys()]);
133     return [CommitShard, root, conflicts, commit];
134
135
136 upon <pl.Deliver | β, [Witness, root, certificate]>:
137     response = handle_witness(root, certificate);
138
139     if response != ⊥:
140         trigger <pl.Send | β, response>;
141
142
143 procedure handle_commit(root, patches):
144     if root not in batches:
145         return ⊥;
146
147     Batch {ids, payloads, ..} = batches[root];
148
149     if exists id in ids such that id not in dir:
150         return ⊥;
151
152     signers: {Server} = {};
153
154     for (conflicts, certificate) in patches:
155         if not certificate.verify([Commit, root, conflicts]):
156             return ⊥;
157
158         signers.extend(certificate.signers());
159
160     if |signers| < 2f + 1:
161         return ⊥;
162
163     exclusions = union(patches.keys());
164     commits[(root, exclusions)] = patches;
165
166     for (id, context, message) in join(ids, payloads):
167         keycard = dir[id];
168
169         if id not in exclusions
              and (keycard, context) not in delivered:
170             delivered.add((keycard, context));
171             trigger <sr.Deliver| keycard, context, message}>;
```

```
172
173     trigger <timer.Set | [OfferTotality, root, exclusions], 7>;
174
175     shard = multisign([Completion, root, exclusions]);
176     return [CompletionShard, root, shard];
177
178
179 upon <pl.Deliver | π, [Commit, root, patches]>:
180     response = handle_commit(root, patches);
181
182     if response != ⊥ and π in B:
183         trigger <pl.Send | π, response>;
184
185
186 upon <timer.Ring | [OfferTotality, root, exclusions]>:
187     for σ in Σ:
188         trigger <pl.Send | σ, [OfferTotality, root, exclusions]>;
189
190
191 upon <pl.Deliver | σ, [OfferTotality, root, exclusions]>:
192     if (root, exclusions) not in commits:
193         trigger <pl.Send | σ, [AcceptTotality, root, exclusions]>;
194
195
196 upon <pl.Deliver| σ, [AcceptTotality, root, exclusions]>:
197     if let batch = batches[root]
          and let patches = commits[(root, exclusions)]:
198         Batch {ids, payloads, ..} = batch;
199
200         assignments = dir.export(ids..);
201         compressed_ids = compress(ids);
202
203         trigger <pl.Send| σ, [Totality, root, assignments,
              (compressed_ids, payloads), patches]>;
204
205
206 upon <pl.Deliver | σ, [Totality, root, assignments, (ids, payloads),
      patches]>:
207         dir.import(assignments..);
208
209         handle_batch(ids, payloads);
210         handle_commit(root, patches)
```

6.3 Correctness

In this section, we prove to the fullest extent of formal detail that Draft implements a Client-Server Byzantine Reliable Broadcast system.

6.3.1 No duplication

In this section, we prove that Draft satisfies no duplication.

Lemma 49. *Let σ be a correct server, let χ be a client, let c be a context. If $(\chi, c) \notin delivered$ at σ, then $server$ did not deliver a message from χ for c.*

Proof. Upon initialization, $delivered$ is empty at σ (line 22). Moreover, σ adds (χ, c) to $delivered$ only by executing line 170. Immediately after doing so, σ delivers a message from χ for c (line 171), and because σ never removes elements from $delivered$, the lemma is proved. □

Theorem 15. *Draft satisfies no duplication.*

Proof. Let σ be a correct server, let χ be a client, let c be a context. σ delivers a message for c from χ only by executing line 171. σ does so only if $(\chi, c) \notin delivered$ (line 169). By Lemma 49, we then have that σ never delivers a message for c from χ more than once, and the theorem is proved. □

6.3.2 Consistency

Notation 16 (Contexts, messages)**.** We use $\mathbb{C}$ and $\mathbb{M}$ to respectively denote the set of contexts and messages broadcast in a Client-Server Byzantine Reliable Broadcast system.

Definition 57 (Ideal accumulator)**.** d An **ideal accumulator** is a tuple $(\mathscr{U}, \mathscr{R}, \mathscr{P}, \rho, \zeta, \nu)$ composed by

- An **universe** $\mathscr{U}$;
- A set of **roots** $\mathscr{R}$;
- A set of **proofs** $\mathscr{P}$;
- A **root function** $\rho : \mathscr{U}^{<\infty} \to \mathscr{R}$;
- A **proof function** $\zeta : \mathscr{U}^{<\infty} \times \mathbb{N} \to \mathscr{P}$;
- A **verification function** $\nu : \mathscr{R} \times \mathscr{P} \times \mathbb{N} \times \mathscr{U} \to \{\texttt{True}, \texttt{False}\}$;

such that

$$\forall z \in \mathcal{U}^{<\infty}, \forall n \leq |z|,\ v\big(\rho(z), \zeta(z,n), n, z_n\big) = \texttt{True}$$
$$\forall z \in \mathcal{U}^{<\infty}, \forall n \leq |z|, \forall q \neq z_n, \forall p \in \mathcal{P},\ v\big(\rho(z), p, n, q\big) = \texttt{False}$$

Let $r \in \mathcal{R}$, let $p \in \mathcal{P}$, let $n \in \mathbb{N}$, let $x \in \mathcal{U}$ such that $v\big(r, p, n, x\big) = \texttt{True}$. We say that p is a proof for x from r.

Lemma 50. *Let $\big(\mathcal{U}, \mathcal{R}, \mathcal{P}, \rho, \zeta, v\big)$ be an ideal accumulator. We have that ρ is injective.*

Proof. Let us assume by contradiction that $z, z' \in \mathcal{U}^{<\infty}$ exist such that $z \neq z'$ and $\rho(z) = \rho\big(z'\big)$. Let $n \in \mathbb{N}$ such that $z_n \neq z'_n$. By Definition 57 we have

$$\begin{aligned}\texttt{True} = v\big(\rho(z), \zeta(z,n), n, z_n\big) = \\ = v\big(\rho\big(z'\big), \zeta(z,n), n, z_n\big) = \texttt{False}\end{aligned}$$

which immediately proves the lemma. □

Throughout the remainder of this document, we use Merkle hash-trees as a cryptographic approximation for an ideal accumulator

$$\big((\mathbb{I} \times \mathbb{C} \times \mathbb{M}), \mathcal{R}, \mathcal{P}, \rho, \zeta, v\big)$$

on the universe of triplets containing an id, a context and a message. In brief: while a successfully verifiable, corrupted proof can theoretically be forged for a Merkle tree (Merkle tree roots are highly non-injective), doing so is unfeasible for a computationally bounded adversary. An in-depth discussion of Merkle trees is beyond the scope of this Part.

Lemma 51. *Let σ be a correct server. Let $(r, b) \in batches$ at σ, let*

$$i = b.ids$$
$$p = b.payloads$$

We have $|i| = \big|p\big|$.

Proof. We start by noting that, upon initialization, $batches$ is empty at σ (line 17). Moreover, σ adds $Batch\{ids: i, payloads: p, ..\}$ to $batches$ only by executing line 60: σ does so only if $|i| = \big|p\big|$ (lines 51 and 52). □

Lemma 52. *Let σ be a correct server. Let $(r, b) \in batches$ at σ, let*

$$i = b.ids$$

The elements of i are all distinct.

Proof. The proof of this lemma is identical to that of Lemma 51, and we omit it for the sake of brevity. □

Definition 58 (Join function)**.** The **join function**

$$\mathtt{j} : \{(i, p) \mid i \in \mathbb{I}^{<\infty}, p \in (\mathbb{C} \times \mathbb{M})^{<\infty}, |i| = |p|\} \to (\mathbb{I} \times \mathbb{C} \times \mathbb{M})^{<\infty}$$

is defined by

$$|\mathtt{j}(i, p)| = (|i| = |p|)$$
$$\mathtt{j}(i, p)_n = (i_n, c_n, m_n), \text{ where } (c_n, m_n) = p_n$$

Lemma 53. *Let σ be a correct server. Let $(r, b) \in batches$ at σ, let $l = \mathtt{j}(b.ids, b.payloads)$. We have $r = \rho(l)$.*

Proof. We underline that, because by Lemma 51 we have $|b.ids| = |b.payloads|$, the definition of l is well-formed. We start by noting that, upon initialization, $batches$ is empty at σ (line 17). Moreover, σ sets

$$batches[r] = Batch\{ids : i, payloads : p, ..\}$$

only by executing line 60. Immediately before doing so, σ computes $r = \rho(l)$ (lines 56, 57 and 58). □

Lemma 54. *Let σ, σ' be correct servers. Let $(r, b) \in batches$ at σ, let $(r, b') \in batches$ at σ'. We have $b = b'$.*

Proof. Upon initialization, $batches$ is empty at both σ and σ'. Moreover, σ and σ' respectively set

$$batches[r] = Batch\{ids : i, payloads : p, tree : t\}$$
$$batches[r] = Batch\{ids : i', payloads : p', tree : t'\}$$

only by executing line 60. By Lemma 53, we have

$$\rho(\mathtt{j}(i, p)) = r = \rho(\mathtt{j}(i', p'))$$

which, by Lemma 50 and the injectiveness of $\mathtt{j}$, proves $i = i'$ and $p = p'$. Because σ (resp., σ') computes t (resp. t') as a pure function of i and p (resp., $i' = i$ and $p' = p$) (line 57), we have $t = t'$. This proves $b = b'$ and concludes the lemma. □

Lemma 55. *Let σ be a correct server, let i be an id. Whenever σ invokes $dir[i]$, σ knows i.*

Proof. σ invokes $dir[i]$ only by executing lines 88, 93 or 167. If σ invokes $dir[i]$ by executing lines 88 or 93, then $i \in I$ (lines 83 and 84, line 91 respectively) for some $I \subseteq \mathbb{I}$ such that, for all $i' \in I$, σ knows i' (lines 79 and 80). Similarly, if σ invokes $dir[i]$ by executing line 167, then $i \in I$ (line 166) for some $I \subseteq \mathbb{I}$ such, for all $i' \in I$, σ knows i' (lines 149 and 150). □

Lemma 56. *Let σ be a correct server. Let χ be a client, let c be a context, let m be a message such that σ delivers (χ, c, m). We have that σ knows χ.*

Proof. σ delivers (χ, c, m) only by executing line 171. Immediately before doing so, σ invokes $dir[i]$ for some $i \in \mathbb{I}$ to obtain χ (line 167). By Lemma 55 we then have that σ knows i which, by Definition 66, proves that σ knows k as well, and concludes the lemma. □

Lemma 57. *Let σ be a correct server. Let χ be a client, let c be a context, let m be a message such that σ delivers (χ, c, m). Let $i = \mathtt{D}(\chi)$. Some $l \in (\mathbb{I} \times \mathbb{C} \times \mathbb{M})^{<\infty}$, $\sigma_1, \ldots, \sigma_{2f+1} \in \Sigma$, $\epsilon_1, \ldots, \epsilon_{2f+1} \subseteq \mathbb{I}$ exist such that:*

- *For some k, $l_k = (i, c, m)$;*
- *For all n, $i \notin \epsilon_n$;*
- *For all n, σ_n signed $[\mathtt{Commit}, \rho(l), \epsilon_n]$.*

Proof. We underline the soundness of the lemma's statement. Because σ delivers $(\chi, ..)$, by Lemma 56 σ knows χ, and by Notation 30 $\mathtt{D}(\chi)$ is well-defined.

We start by noting that some $b = batches[r]$ exists (lines 144, 145 and 147) such that, for some k, we have $i = b.ids[k]$ and $(c, m) = b.payloads[k]$ (line 166). Let $l = \mathtt{j}(b.ids, b.payloads)$. By Definition 58 we immediately have $l_k = (i, c, m)$. Moreover, by Lemma 53, we have $r = \rho(l)$.

Moreover, some $S \subseteq \Sigma$ exists such that:

- $|S| \geq 2f + 1$ (lines 160 and 161);
- For all $\sigma' \in S$, some $\epsilon(\sigma')$ exists such that σ' signed $[\mathtt{Commit}, r, \epsilon(\sigma')]$ (lines 152, 155, 156 and 158: S is obtained by iteratively extending an initially empty set with certificate signers);

Let $\sigma_1, \ldots, \sigma_{2f+1}$ be distinct elements of S, let $\epsilon_n = \epsilon(\sigma_n)$.

Finally, some $\epsilon \supseteq \epsilon_1 \cup \ldots \cup \epsilon_{2f+1}$ exists (lines 154 and 163) such that $i \notin \epsilon$ (line 169). For all n we then obviously have $i \notin \epsilon_n$.

In summary: $l_k = (i, c, m)$; for all n, σ_n signed $[\mathtt{Commit}, \rho(l) = r, \epsilon_n]$; for all n, $i \notin \epsilon_n$. The lemma is therefore proved. □

Lemma 58. *Let σ be a correct server, let i be an id, let c be a context, let m be a message. Let $t \in \mathbb{R}$ such that, at time t, $messages[(i,c)] = (m,_)$. At any time $t' > t$, we have $messages[(i,c)] = (m,_)$ as well.*

Proof. After setting $messages[(i,c)] = (m,_)$, σ sets $messages[(i,c)] = (m,_)'$ only by executing line 130. It does so only if $m = m'$ (line 120). □

Lemma 59. *Let σ be a correct server. Let i be an id, let c be a context, let m be a message. Let $l \in (\mathbb{I} \times \mathbb{C} \times \mathbb{M})^{<\infty}$, let $\epsilon \subseteq \mathbb{I}$ such that:*

- *For some k, $l_k = (i,c,m)$;*
- *$i \notin \epsilon$;*
- *σ signs $[\mathtt{Commit}, \rho(l), \epsilon]$.*

We have that σ sets $messages[(i,c)] = (m,_)$.

Proof. Prior to signing $[\mathtt{Commit}, \rho(l), \epsilon]$ (line 132), σ retrieves $b' = batches[\rho(l)]$ (lines 108 and 109 and 111). Let $l' = \mathtt{j}(b'.ids, b'.payloads)$. By Lemma 53 we have $\rho(l') = \rho(l)$ which, by Lemma 50, proves $l = l'$.

Subsequently, σ loops through all elements of $l = l'$ (line 119). For each (i_j, c_j, m_j) in l, σ either adds i_j to ϵ (line 128), or sets $messages[i_j, c_j] = (m_j,_)$ (line 130). Because σ does so in particular for $j = k$, and $i \notin \epsilon$, σ sets $messages[(i,c)] = (m,_)$, and the lemma is proved. □

Theorem 16. *Draft satisfies consistency.*

Proof. Let σ, σ' be two correct servers. Let χ be a client, let c be a context, let m, m' be messages such that σ and σ' deliver (χ, c, m) and (χ, c, m') respectively. By Lemma 56, σ knows χ. Let $i = \mathtt{D}(\chi)$. By Lemma 57, and noting that at most f servers are Byzantine, some $l \in (\mathbb{I} \times \mathbb{C} \times \mathbb{M})^{<\infty}$, $\sigma_1, \ldots, \sigma_{f+1} \in \Sigma$, and $\epsilon_1, \ldots, \epsilon_{f+1}$ exist such that: for some k, $l_k = (i,c,m)$; for all n, σ_n is correct; for all n, $i \notin \epsilon_n$; for all n, σ_n signed $[\mathtt{Commit}, \rho(l), \epsilon_n]$. By Lemmas 59 and 58, $messages[(i,c)]$ is permanently set to $(m,_)$ at $\sigma_1, \ldots, \sigma_{f+1}$.

By an identical reasoning, some $\sigma'_1, \ldots, \sigma'_{f+1}$ exist such that for all n, σ'_n is correct and permanently sets $messages[(i,c)] = (m',_)$. Because $(\sigma_1, \ldots, \sigma_{f+1})$ and $(\sigma'_1, \ldots, \sigma'_{f+1})$ intersect in at least one server, we have $m = m'$, and the theorem is proved. □

6.3.3 Totality

Notation 17 (Ordering)**.** Let X be a set endowed with a total order relationship. We use $X^{\nearrow} \subset X^{<\infty}$ to denote the set of finite, non-decreasing sequences on X. Let $Y \subseteq X$. We use $\mathscr{S}(Y)$ to denote **sorted** Y, i.e., the sequence containing all elements of Y in ascending order.

Definition 59 (Id compression and expansion)**.** The **id compression function**

$$\mathsf{c} : \mathbb{I}^{\nearrow} \to (\mathbb{D} \to \mathbb{P}(\mathbb{N}))$$

is defined by

$$n \in \mathsf{c}(i)(d) \Longleftrightarrow \exists j \mid i_j = (d, n)$$

The **id expansion function**

$$\mathsf{e} : (\mathbb{D} \to \mathbb{P}(\mathbb{N})) \to \mathbb{I}^{\nearrow}$$

is defined by

$$\mathsf{e}\big(f\big) = \mathscr{S}\big(\big\{(d, n) \mid n \in f(d)\big\}\big)$$

Lemma 60. *Let $i \in \mathbb{I}^{\nearrow}$. We have $\mathsf{e}(\mathsf{c}(i)) = i$.*

Proof. It follows immediately from Definition 59. □

Lemma 61. *Let σ, σ' be correct processes, let $(r, b) \in batches$ at σ, let*

$$i = b.ids$$
$$p = b.payloads$$

Upon evaluating $handle_batch\big(\mathsf{c}(i), p\big)$, σ' sets $batches[r] = b$.

Proof. We start by noting that, upon initialization, $batches$ is empty at σ (line 17). Moreover σ adds (r, b) to $batches$ only by executing line 60. σ does so only if i has no duplicates, and $|i| = \big|p\big|$ (line 51).

Upon invoking $handle_batch\big(\mathsf{c}(i), p\big)$, σ' verifies that $i = \mathsf{e}(\mathsf{c}(i))$ (line 49, see Lemma 60) has no duplicates and satisfies $|i| = \big|p\big|$ (line 51). Having done so, σ' sets $batches[r] = b'$ for some b'. However, by Lemma 54 we have $b = b'$, and the lemma is proved.

□

Lemma 62. *Let σ, σ' be correct processes, let $((r, \epsilon), \zeta) \in commits$ at σ, let $(r, b) \in batches$ at σ' such that for all $i \in b.ids$, σ' knows i. Upon evaluating $handle_commit(r, \zeta)$, σ' sets $commits[(r, \epsilon)] = \zeta$.*

Proof. We start by noting that, upon initialization, $commits$ is empty at σ (lines 19 and 156). Moreover σ adds $((r, \epsilon), \zeta)$ to $commits$ only by executing line 164. σ does so only if, for all

$(c, z) \in \zeta$ (line 154), z verifies correctly against $[\texttt{Commit}, r, c]$ (line 155) and

$$\left| \bigcup_{(_, z) \in \zeta} z.signers() \right| \geq 2f + 1$$

(lines 160 and 161). We additionally have

$$\epsilon = \bigcup_{(c, _) \in \zeta} c$$

(line 163).

Upon invoking $handle_commits(r, \zeta)$, by hypothesis σ' verifies that $r \in batches$ (line 144) and, for all $i \in b.ids$ (line 147), $i \in dir$ (line 149). Moreover, because $verify$ is a pure procedure, σ' passes the checks at lines 155 and 160. Next, σ' computes

$$\epsilon' = \bigcup_{(c, _) \in \zeta} c = \epsilon$$

(line 163) and sets $commits[(r, \epsilon')] = \zeta$ (line 164), concluding the lemma. □

Lemma 63. *Let σ be a correct process, let $(r, _) \in commits$ at σ. We have $r \in batches$ at σ.*

Proof. Upon initialization, $commits$ is empty at σ (line 19). Moreover, σ adds $(r, _)$ to $commits$ only by executing line 164. σ does so only if $r \in batches$ (lines 144 and 145). □

Lemma 64. *Let σ be a correct process, let $(r, _) \in commits$ at σ. Let $b = batches[r]$ at server. For all $i \in b.ids$, σ knows i.*

Proof. We start by noting that, by Lemma 63, the definition of b is well-formed. Upon initialization, $commits$ is empty at σ (line 19). Moreover, σ adds $(r, _)$ to $commits$ only by executing line 164. σ does so only if, for all i in $b.ids$, σ knows i (lines 147 and 149 and 150). □

Lemma 65. *Let σ be a correct process, let $(r, \epsilon) \in commits$ at σ. Let $b = batches[r]$ at σ, let*

$$i = b.ids$$
$$p = b.payloads$$

For all j, let $(c_j, _) = p_j$. For all j such that $i_j \notin \epsilon$, σ delivered a message from $\texttt{D}(i_j)$ for c_j.

Proof. We underline that, by Lemma 64, $\texttt{D}(i_j)$ is well-defined for all j. Upon initialization, $commits$ is empty at σ (line 19). Moreover, σ adds (r, ϵ) to $commits$ only by executing line 164. Upon doing so, σ retrieves b from $batches$ (line 147) and delivers a message from $\texttt{D}(i_j)$ for c_j (lines 167 and 171) for all j such that: $i_j \notin \epsilon$; and σ did not previously deliver a message from $\texttt{D}(i_j)$ for c_j (line 169, see Lemma 49). □

Theorem 17. *Draft satisfies totality.*

Proof. Let σ, σ' be correct servers. Let χ be a client, let c be a context such that σ delivers a message from χ for c. σ delivers a message from χ for c only by executing line 171. Immediately before doing so, σ sets $commits[r, \epsilon] = \zeta$ for some r, ϵ and ζ such that $\mathtt{D}(\chi) \notin \epsilon$ (lines 164, 167 and 169). Let $b = batches[r]$ at σ (by Lemma 63, $r \in batches$ at σ), let

$$i = b.ids$$
$$p = b.payloads$$

For all j, let $(c_j, _) = p_j$. For some k we have $i_k = \mathtt{D}(\chi)$ and $c_k = c$ (lines 147, 166 and 167).

Immediately after delivering a message from χ for c, σ sets a timer for $[\mathtt{OfferTotality}, r, \epsilon]$ (line 173). When the timer eventually rings (line 186), σ sends an $[\mathtt{OfferTotality}, r, \epsilon]$ message to all servers, including σ' (lines 187 and 188). Upon eventually delivering $[\mathtt{OfferTotality}, r, \epsilon]$ (line 191), σ' checks if $(r, \epsilon) \in commits$ (line 192). Let us assume that $(r, \epsilon) \in commits$ at σ'. Let $b' = batches[r]$ at σ' (by Lemma 63, $r \in batches$ at σ'). By Lemma 54, we have $b' = b$. As a result, by Lemma 65, σ' already delivered a message from $\mathtt{D}(i_k) = \chi$ for $c_k = c$. Throughout the remainder of this proof, we assume $(r, \epsilon) \notin commits$ at σ'.

Upon verifying that $(r, \epsilon) \notin commits$ (line 192), σ' sends back to σ an $[\mathtt{AcceptTotality}, r, \epsilon]$ message (line 193). Upon delivering $[\mathtt{AcceptTotality}, r, \epsilon]$ (line 196), σ verifies that $r \in batches$ and $(r, \epsilon) \in commits$ (line 197), exports an array a of assignments for all elements of i (line 200) (by Lemma 64 σ knows all elements of i), and sends $[\mathtt{Totality}, r, a, (\mathtt{c}(i), p), \zeta]$ to σ' (line 203)

Upon delivering $[\mathtt{Totality}, r, a, (\mathtt{c}(i), p), \zeta]$ (line 206), σ' imports all elements of a (line 207). Having done so, σ' knows all elements of i. Then, σ' invokes $handle_batch(\mathtt{c}(i), p)$ (line 209). By Lemma 61, having done so σ' sets $batches[r] = b$. Finally, σ' invokes $handle_commit(r, \zeta)$ (line 210). By Lemma 62, having done so σ' sets $commits[(r, \epsilon)] = \zeta$. By Lemma 65, σ' delivers a message from $\mathtt{D}(\chi) = i_k$ for $c = c_k$.

In summary, if σ delivers a message from χ for c, then σ' eventually delivers a message from χ for c as well, and the theorem is proved. □

6.3.4 Integrity

In this section, we prove that Draft satisfies integrity.

Lemma 66. *Let σ be a correct server. Let χ be a client, let c be a context, let m be a message such that σ delivers m from χ for c. Some correct server σ' and some $(r, b) \in batches$ at σ' exists such*

that σ' *signed* $[\texttt{Witness}, r]$ *and, with*

$$i = b.ids$$
$$p = b.payloads$$

some k *exists such that* $i_k = \mathtt{D}(\chi)$ *and* $p_k = (c, m)$.

Proof. We start by noting that σ delivers m from χ for c only by executing line 171. When σ does so, some $(r, b) \in batches$ exists at σ (line 147) such that, with

$$i = b.ids$$
$$p = b.payloads$$

some k exists such that $i_k = \mathtt{D}(\chi)$ and $p_k = (c, m)$ (lines 166 and 167). Moreover, a set $S \subseteq \Sigma$ exists such that $|S| \geq 2f + 1$ (lines 160 and 161) and, for all $\tilde{\sigma} \in S$, $\tilde{\sigma}$ signed a $[\texttt{Commit}, r, _]$ message (lines 152, 155, 156 and 158). Noting that at most f processes are Byzantine, at least one element σ^* of S is correct.

σ^* signs $[\texttt{Commit}, r, _]$ only upon executing line 132. σ^* does so only if at least $f + 1$ servers signed $[\texttt{Witness}, r]$ (lines 113 and 114). Noting that at most f processes are Byzantine, at least one correct server σ' signed $[\texttt{Witness}, r]$.

σ' signs $[\texttt{Witness}, r]$ only by executing line 96. It does so only if some $(r, b') \in batches$ exists at σ' (lines 74, 75). By Lemma 54, we have $b' = b$.

In summary, some correct server σ' exists such that σ' signed $[\texttt{Witness}, r]$ and $(r, b) \in batches$ at σ', with $i_k = \mathtt{D}(\chi)$ and $p_k = (c, m)$: the lemma is proved. □

Lemma 67. *Let* χ *be a correct client. Let* $(c, s) \in submissions$ *at* χ, *let*

$$m = s.message$$

We have that χ *broadcast* m *for* c.

Proof. Upon initialization, $submissions$ is empty at χ (line 25); χ adds (c, s) to $submissions$ (line 32) only upon broadcasting m for c (line 29). □

Lemma 68. *Let* σ *be a correct server, let* $l \in (\mathbb{I} \times \mathbb{C} \times \mathbb{M})^{\nearrow}$ *such that* σ *signs* $[\texttt{Witness}, \rho(l)]$. *For all* j, *let* $(i_j, c_j, m_j) = l_j$, *let* $\chi_j = \mathtt{D}(i_j)$. *For all* k *such that* χ_k *is correct,* χ_k *broadcast* m_k *for* c_k.

Proof. We start by noting that σ signs $[\texttt{Witness}, \rho(l)]$ only by executing line 96. Immediately before doing so, σ loads a batch $b = batches[\rho(l)]$ (line 77). By Lemmas 53 and 50, we have

$$l = \mathtt{j}(b.ids, b.payloads)$$

Moreover, by Lemma 52, all elements of i are distinct.

Next, σ uses a set $H \subseteq \mathbb{I}$ to partition the elements of i. For each $h \in H$ (line 82), σ verifies that: for some j, we have $h = i_j$ (lines 83 and 89); and χ_j signed $[\texttt{Message}, c_j, m_j]$ (lines 86, 88 and 89). Finally, for all j such that $i_j \notin H$ (line 91), σ verifies that χ_j signed $[\texttt{Reduction}, \rho(l)]$ (lines 93 and 94).

In summary, for all j, we have that either χ_j signed $[\texttt{Message}, c_j, m_j]$, or χ_j signed $[\texttt{Reduction}, \rho(l)]$.

Let k such that χ_k is correct. Let us assume that χ_k signed $[\texttt{Message}, c_k, m_k]$. χ_k signs $[\texttt{Message}, c_k, m_k]$ only by executing line 30. It does so only upon broadcasting m_k for c_k (line 29). Throughout the remainder of this proof, we assume that χ_k signed $[\texttt{Reduction}, \rho(l)]$.

χ_k signs $[\texttt{Reduction}, \rho(l)]$ only by executing line 55. It does so only if, for some $p \in \mathscr{P}$, $n \in \mathbb{N}$, $c' \in \mathbb{C}$, $m' \in \mathbb{M}$, we have

$$\nu\big(\rho(l), p, n, (i_k, c', m')\big) = \texttt{True} \tag{6.1}$$

(line 52), and χ_k previously broadcast m' for c' (line 51, see Lemma 67). By Equation 6.1 and Definition 57, we must have $l_n = (i_k, \ldots)$. As a result, because i is non-repeating, we immediately have $n = k$, $c' = c_k$ and $m' = m_k$. We then have that χ_k previously broadcast m_k for c_k, and the lemma is proved. □

Theorem 18. *Draft satisfies integrity.*

Proof. Let σ be a correct server. Let χ be a correct client, let c be a context, let m be a message such that σ delivers m from χ for c. By Lemma 66, some correct server σ' and some $(r, b) \in batches$ at σ' exist such that σ' signed $[\texttt{Witness}, r]$ and, with

$$i = b.ids$$
$$p = b.payloads$$

some k exists such that $i_k = \texttt{D}(\chi)$ and $p_k = (c, m)$. Let $l = \texttt{j}(i, p)$. We trivially have $l_k = (\texttt{D}(\chi), c, m)$. By Lemma 53 we have $r = \rho(l)$. For all j, let $(c_j, m_j) = p_j$. By Lemma 68, we have that $(\chi = \texttt{D}(\texttt{D}(\chi)) = \texttt{D}(i_k))$ broadcast $(m = m_k)$ for $(c = c_k)$, and the theorem is proved.

□

6.3.5 Validity

In this section, we prove that Draft satisfies validity.

Lemma 69. *Let χ be a correct client. Upon initialization, χ knows χ.*

Proof. It follows immediately from lines 22 and 23, and the signup validity of Directory. □

Lemma 70. *Let χ be a correct client, let c be a context such that χ broadcast a message for c. If $c \notin submissions$ at χ, then at least one correct server delivered a message from χ for c.*

Proof. Let $i = \mathtt{D}(\chi)$ (by Lemma 69, i is well-defined). Let m be the message χ broadcast for c. We start by noting that χ adds c to $submissions$ (line 32) upon broadcasting a message for c (line 29). As a result, χ satisfies $c \notin submissions$ only upon removing (c, s) from $submissions$ (line 64 only). χ does so only if

$$s.included_in \cap completed \neq \emptyset$$

Let $r \in s.included_in \cap completed$.

Noting that $s.included_in$ is initially empty at χ (line 32), χ must have added r to $s.included_in$ (line 53 only). Noting that χ permanently sets $s.message$ to m upon initialization (line 32 only, see line 29), χ adds r to $s.included_in$ (line 53) only if, for some $p \in \mathscr{P}$ and $n \in \mathbb{N}$, we have

$$v\big(r, p, n, (i, c, m)\big) = \mathtt{True} \tag{6.2}$$

(line 52).

Noting that $completed$ is initially empty at χ (line 26), χ must have added r to $completed$ (line 60 only). χ does so only if some ϵ exists such that $i \notin \epsilon$ and at least $f + 1$ servers signed $\big[\mathtt{Completion}, r, \epsilon\big]$ (line 59).

Noting that at most f servers are Byzantine, some correct server σ exists such that σ signed $\big[\mathtt{Completion}, r, \epsilon\big]$ (line 175 only). σ does so only if $r \in batches$ (lines 144 and 145). Let $b = batches[r]$ at σ, let $l = \mathtt{j}\big(b.ids, b.payloads\big)$, let $\big(i_j, c_j, m_j\big) = l_j$, let $\chi_j = \mathtt{D}\big(i_j\big)$. By Lemma 53, we have $r = \rho(l)$, and by Equation 6.2 and Definition 57 we have $l_n = (i, c, m)$.

Immediately before signing $\big[\mathtt{Completion}, r, \epsilon\big]$, σ loops through l (line 166) and delivers m_j from χ_j for c_j (lines 167 and 171) if and only if $i_j \notin \epsilon$ and σ did not previously deliver a message from χ_j for c_j (line 169, see Lemma 49). Recalling that $(i = i_n) \notin \epsilon$, σ delivers a message from $\big(\chi = \chi_j\big)$ for $\big(c = c_j\big)$ before or upon signing $\big[\mathtt{Completion}, r, \epsilon\big]$, and the lemma is proved. □

Lemma 71. *Let χ be a correct client, let c be a context, let m be a message such that χ broadcast m for c. Eventually, either a correct server delivers a message from χ for c, or a correct broker delivers a $[Submission, a, (c, m, s)]$ message, where a is an assignment for χ and s is χ's signature for $\big[Message, c, m\big]$.*

Proof. Upon broadcasting m for c (line 29), χ sets

$$submissions[c] = Submission\big\{message : m, signature : s, submitted_to : \emptyset, ..\big\}$$

where s is a signature for $[\texttt{Message}, c, m]$ (lines 30 and 32). χ never updates $submissions[c].message$ or $submissions[c].signature$. Subsequently, χ invokes $submit(c)$ (line 34).

Upon executing $submit(c)$ (line 37), χ immediately returns if and only if $c \notin submissions$, or no $\beta \in (B \setminus submissions[c].submitted_to)$ exists (line 38). Otherwise, χ sends a $[\texttt{Submission}, a, (c, m, s)]$ message to β (where a is an assignment for χ) (lines 39 and 40), then adds β to $submissions[c].submitted_to$ (line 42), and finally schedules $submit(c)$ for eventual re-execution (lines 43, 46 and 47).

Noting that χ updates $submissions[c].submitted_to$ only by executing line 42, χ keeps re-executing $submit(c)$ until either

- $c \notin submissions$: by Lemma 70, at least one correct server delivered a message from χ for c; or

- χ sent to all brokers a $[\texttt{Submission}, a, (c, m, s)]$ message, where a is an assignment for χ

The lemma follows immediately from the observation that B is finite and the assumption that at least one broker is correct. □

Lemma 72. *Let β be a correct broker. If $pool \neq \{\}$ at β, then eventually $pool = \{\}$ at β.*

Proof. We start by noting that the $[\texttt{Flush}]$ timer is pending at β if and only if $collecting = \texttt{True}$ at β. Indeed, upon initialization, we have $collecting = \texttt{False}$ at β. Moreover, β sets $collecting = \texttt{True}$ (line 63) only upon setting the $[\texttt{Flush}]$ timer (line 64 only). Finally, β resets $collecting = \texttt{False}$ (line 68 only) only upon ringing $[\texttt{Flush}]$ (line 67).

Upon ringing $[\texttt{Flush}]$ (line 67), β resets $pool = \{\}$ (line 71). As a result, whenever $pool \neq \{\}$ at β, we either have:

- $collecting = \texttt{True}$ at β: $[\texttt{Flush}]$ is pending at β and β will eventually reset $pool = \{\}$; or

- $collecting = \texttt{False}$ at β: β will eventually detect that $pool \neq \{\}$ and $collecting = \texttt{False}$ (line 62) and set $collecting = true$ - the above will then apply.

□

Lemma 73. *Let β be a correct broker, let i be an id such that*

$$pending[i] = [s_1, \ldots, s_S]$$

at β, with $S \geq 1$. We eventually have

$$pool[i] = s_1$$
$$pending[i] = [s_2, \ldots, s_S, \ldots]$$

at β.

Proof. We start by noting that, by Lemma 72, if $i \in pool$ then eventually $pool = \{\}$ at β. Noting that β adds elements to $pool$ only by executing line 59, we then have that eventually β detects $pending[i] \neq []$ and $i \notin pool$ (line 57). Upon doing so, β sets $pending[i] = [s_2, \ldots, s_S, \ldots]$ (line 58) and $pool[i] = s_1$ (line 59). □

Lemma 74. *Let β be a correct broker, let i be an id, let s be a submission such that β pushes s to $pending[i]$. We eventually have $pool[i] = s$ at β.*

Proof. It follows immediately by induction on Lemma 73, and the observation that β only pushes elements to the back of $pending$. □

Lemma 75. *Let β be a correct broker. Let χ be a client, let a be an assignment for χ, let c be a context, let m be a message, let s be χ's signature for $[\texttt{Message}, c, m]$. If β delivers a $[\texttt{Submission}, a, (c, m, s)]$, then β eventually sets*

$$pool[i] = Submission\{context : c, message : m, ..\}$$

where $i = \mathrm{D}(\chi)$.

Proof. Upon delivering $[\texttt{Submission}, a, (c, m, s)]$ (line 50), β imports a (line 51). Because s successfully verifies against $[\texttt{Message}, c, m]$ and β knows χ (line 53), β pushes $Submission\{context : c, message : m, ..\}$ to $pending[i]$ (line 54). The lemma immediately follows from Lemma 74. □

Definition 60 (Map)**.** Let X, Y be sets. A **map** $m : X \rightarrowtail Y$ is a subset of $(X \times Y)$ such that

$$\forall (x, y), (x', y') \neq (x, y) \in m, x \neq x'$$

We use

$$\mathscr{D}(m) = \{x \in X \mid (x, _) \in m\}$$
$$\mathscr{C}(m) = \{y \in Y \mid (_, y) \in m\}$$

to respectively denote the **domain** and **codomain** of m.

Notation 18 (Sorting maps)**.** Let X, Y be sets such that X is endowed with a total order relationship, let $m : X \rightarrowtail Y$. With a slight abuse of notation, we use $\mathscr{S}(\mathscr{C}(m))$ to denote the

codomain of m, sorted by m's domain. For example, let $m = \{(1, \square), (2, \circ), (3, \triangle)\}$, we have $\mathscr{S}(\mathscr{C}(m)) = (\square, \circ, \triangle)$.

Definition 61 (Broker variables)**.** Let β be a correct broker, let $t \in \mathbb{R}$, let r be a root.

- We use $\mathtt{P}_\beta(t)[r] : \mathbb{I} \rightarrowtail (\mathbb{M} \times \mathbb{C})$ to denote, if it exists, the value of $batches[r].payloads$ at β at time t; we use $\mathtt{P}_\beta(t)[r] = \perp$ otherwise.
- We define $\mathtt{C}_\beta(t)[r] : \mathbb{I} \rightarrowtail \mathbb{C}$ and $\mathtt{M}_\beta(t)[r] : \mathbb{I} \rightarrowtail \mathbb{M}$ by

  $$\left(\mathtt{C}_\beta(t)[r][i], \mathtt{M}_\beta(t)[r][i]\right) = \mathtt{P}_\beta(t)[r][i]$$

  if $\mathtt{P}_\beta(t)[r][i] \neq \perp$; we use $\mathtt{C}_\beta(t)[r][i] = \mathtt{M}_\beta(t)[r][i] = \perp$ otherwise.
- We define $\mathtt{L}_\beta(t)[r] \in (\mathbb{I} \times \mathbb{C} \times \mathbb{M})^{\nearrow}$ by

  $$\mathtt{L}_\beta(t)[r] = \mathscr{S}\left(\left\{(i, c, m) \mid (i, (c, m)) \in \mathtt{P}_\beta(t)[r]\right\}\right)$$

  if $\mathtt{P}_\beta(t)[r] \neq \perp$; we use $\mathtt{L}_\beta(t)[r] = \perp$ otherwise.
- We use $\mathtt{S}_\beta(t)[r] : \mathbb{I} \rightarrowtail \mathbb{S}^1$ and $\mathtt{Q}_\beta(t)[r] : \mathbb{I} \rightarrowtail \mathbb{S}^+$ to respectively denote, if they exist, the values of $batches[r].signatures$ and $batches[r].reductions$ at β at time t; we use $\mathtt{S}_\beta(t)[r] = \mathtt{Q}_\beta(t)[r] = \perp$ otherwise.
- We use $\mathtt{Y}_\beta(t)[r] \subseteq \Sigma$ and $\mathtt{Z}_\beta(t)[r] \in \{\mathtt{True}, \mathtt{False}\}$ to respectively denote, if they exist, the values of $batches[r].commit_to$ and $batches[r].committable$ at β at time t; we use $\mathtt{Y}_\beta(t)[r] = \mathtt{Z}_\beta(t)[r] = \perp$ otherwise.
- We use $\mathtt{W}_\beta(t)[r] : \Sigma \rightarrowtail \mathbb{S}^+$ to denote, if it exists, the value of $batches[r].witnesses$ at β at time t; we use $\mathtt{W}_\beta(t)[r] = \perp$ otherwise. We underline that $batches[r].witnesses$ exists only if $batches[r]$ exists and is `Witnessing`.
- We use $\mathtt{X}_\beta(t)[r] : \Sigma \rightarrowtail \left(\mathbb{P}(\mathbb{I}) \times \mathbb{S}^+\right)$ to denote, if it exists, the value of $batches[r].commits$ at β at time t; we use $\mathtt{X}_\beta(t)[r] = \perp$ otherwise. We underline that $batches[r].commits$ exists only if $batches[r]$ exists and is `Committing`.
- We use $\mathtt{E}_\beta(t)[r] \in \mathbb{P}(\mathbb{I})$ and $\mathtt{T}_\beta(t)[r] : \Sigma \rightarrowtail \mathbb{S}^+$ to respectively denote, if they exist, the values of $batches[r].exclusions$ and $batches[r].completions$ at β at time t; we use $\mathtt{E}_\beta(t)[r] = \mathtt{T}_\beta(t)[r] = \perp$ otherwise. We underline that $batches[r].exclusions$ and $batches[r].completions$ exist only if $batches[r]$ exists and is `Completing`.

Notation 19 (Broker variables)**.** Let r be a root. Wherever it can be unequivocally inferred from context, we omit the correct broker and the time from $\mathtt{P}[r]$, $\mathtt{C}[r]$, $\mathtt{M}[r]$, $\mathtt{L}[r]$, $\mathtt{S}[r]$, $\mathtt{Q}[r]$, $\mathtt{Y}[r]$, $\mathtt{Z}[r]$, $\mathtt{W}[r]$, $\mathtt{X}[r]$, $\mathtt{E}[r]$ and $\mathtt{T}[r]$.

Lemma 76. *Let β be a correct broker, let r be a root. We have*

$$(\mathtt{P}[r] = \bot) \iff (\mathtt{C}[r] = \bot) \iff (\mathtt{M}[r] = \bot) \iff$$
$$\iff (\mathtt{Y}[r] = \bot) \iff (\mathtt{Z}[r] = \bot) \iff (\mathtt{L}[r] = \bot) \iff$$
$$\iff (\mathtt{S}[r] = \bot) \iff (\mathtt{Q}[r] = \bot)$$

and

$$(\mathtt{P}[r] = \bot) \implies (\mathtt{W}[r] = \mathtt{X}[r] = \mathtt{E}[r] = \mathtt{T}[r] = \bot)$$

Proof. It follows immediately from Definition 61. □

Lemma 77. *Let β be a correct broker, let i be an id, let c be a context, let m be a message such that*

$$pool[i] = Submission\{context : c, message : m, ..\}$$

at β. Eventually some root r exists such that $\mathtt{P}[r][i] = (c, m)$.

Proof. By Lemma 72, eventually $pool = \{\}$ at β. Moreover, β resets $pool = \{\}$ only by executing line 71. Upon first doing so, β stores the original value of $pool$ in a variable u (line 70), which it uses to define a variable p by

$$\big((i, (c, m)) \in p\big) \iff ((i, Submission\{c, m, ..\}) \in u)$$

(line 83). We immediately have $p[i] = (c, m)$. Finally, for some r, β adds (r, b) to $batches$, with $b.payloads = p$ (line 87). The lemma follows immediately from Definition 61. □

Lemma 78. *Let χ be a correct client, let $i = \mathtt{D}(\chi)$. Let c be a context, let m be a message such that χ broadcast m for c. Eventually, either a correct server delivers a message from χ for c, or some correct broker β and some root r exist such that eventually $\mathtt{P}[r][i] = (c, m)$.*

Proof. The lemma immediately follows from Lemmas 71, 75 and 77. □

Lemma 79. *Let β be a correct broker, let i be an id. If $i \in pool$ at β, then β knows i.*

Proof. We start by noting that, upon initialization, $pool$ is empty at β (line 44). Moreover, β adds $(i, _)$ to $pool$ only by executing line 59. β does so only if $pending[i]$ is not empty (line 57). Upon initialization, all values of $pending$ are also empty at β (line 43). Finally, β adds elements to $pending[i]$ only by executing line 54. Because β does so only if β knows i (line 53), the lemma is proved. □

Lemma 80. *Let β be a correct broker, let r be a root such that $\mathtt{P}[r] \neq \bot$. For all $i \in \mathscr{D}(\mathtt{P}[r])$, β knows i.*

Proof. We start by noting that, upon initialization, $batches$ is empty at β (line 47). Moreover, β adds (r, b) to $batches$ only by executing line 87. Upon doing so, β satisfies $\mathscr{D}(b.payloads) = \mathscr{D}(u)$ (line 83) for some u initialized to $pool$ (line 70). The lemma immediately follows from Lemma 79. □

Lemma 81. *Let β be a correct broker, let r be a root such that* $(\mathtt{P}[r] \neq \bot) \iff (\mathtt{S}[r] \neq \bot)$. *We have* $\mathscr{D}(\mathtt{P}[r]) = \mathscr{D}(\mathtt{S}[r]) \cup \mathscr{D}(\mathtt{Q}[r])$ *and* $\mathscr{D}(\mathtt{S}[r]) \cap \mathscr{D}(\mathtt{Q}[r]) = \emptyset$

Proof. We start by noting that, upon initialization, $batches$ is empty at β (line 47). Moreover, β adds (r, b) to $batches$ only by executing line 87. Upon doing so, β satisfies $\mathscr{D}(b.payloads) = \mathscr{D}(b.signatures)$ (lines 83 and 85). Subsequently, β updates $batches[r].signatures$ and $batches[r].reductions$ only concurrently, by executing lines 95 and 96. Upon doing so, β shifts an id i in $batches[r].payloads$ (line 93) from $\mathscr{D}(batches[r].signatures)$ to $\mathscr{D}(batches[r].reductions)$. □

Lemma 82. *Let β be a correct broker, let r be a root such that* $\mathtt{S}[r] \neq \bot$. *Let* $i \in \mathscr{D}(\mathtt{S}[r])$, *let* $\chi = \mathtt{D}(i)$, *let* $(c, m) = \mathtt{P}[r][i]$, *let* $s = \mathtt{S}[r][i]$. *We have that s is χ's signature for* $[\mathtt{Message}, c, m]$.

Proof. We underline that c and m are well-defined by Lemma 81, and i is well-defined by Lemmas 81 and 80. We start by noting that, upon initialization, $batches$ is empty at β (line 47). Moreover, β adds (r, b) to $batches$ only by executing line 87. Upon last doing so, β satisfies

$$(c, m) = b.payloads[i] = (w.context, w.message)$$
$$s = b.signatures[i] = w.signature$$

with $w = u[i]$ for some copy u of $pool$ (lines 70, 83 and 85).

Upon initialization, $pool$ is empty at β as well (line 44). Moreover, β sets $pool[i] = w$ only by executing line 59. It does so only if w was in $pending[i]$ (lines 57 and 58). Finally, $pending$ is initially empty at β (line 43), and β adds w to $pending[i]$ only by executing line 54. β does so only if $w.signature$ is $\mathtt{D}(i)$'s signature for $[\mathtt{Message}, w.context, w.message]$ (line 53). This proves that s is χ's signature for $[\mathtt{Message}, c, m]$ and concludes the lemma. □

Lemma 83. *Let β be a correct broker, let r be a root such that* $\mathtt{Q}[r] \neq \bot$. *Let* $i \in \mathscr{D}(\mathtt{Q}[r])$, *let* $\chi = \mathtt{D}(i)$, *let* $q = \mathtt{Q}[r][i]$. *We have that q is χ's multisignature for* $[\mathtt{Reduction}, r]$.

Proof. We start by noting that, upon initialization, $batches$ is empty at β (line 47). Moreover, β adds (r, b) to $batches$ only by executing line 87. Upon doing so, β satisfies $b.reductions = \{\}$. Finally, β adds (i, q) to $batches[r].reductions$ only by executing line 95. β does so only if q is $(\chi = \mathtt{D}(i))$'s multisignature for $[\mathtt{Reduction}, r]$ (lines 93 and 94). □

Lemma 84. *Let β be a correct broker, let r be a root such that* $\mathtt{P}[r] \neq \bot$. *We have* $r = \rho(\mathtt{L}[r])$.

Proof. We start by noting that β adds elements to $batches$ only by executing line 87, and β never updates $batches[r].payloads$. Immediately before adding (r, b) to $batches$ (line 87), β computes $r = \rho(\mathscr{S}(l))$ (lines 75 and 76), with

$$\begin{aligned}((i, c, m) \in l) &\iff ((i, Submission\{c, m, ..\}) \in u) \\ &\iff \big((i, (c, m)) \in b.payloads\big)\end{aligned}$$

for some u (lines 73 and line 83). The lemma immediately follows from Definition 61. □

Lemma 85. *Let β be a correct broker, let r be a root, let $t, t' \in \mathbb{R}$ such that $\mathtt{P}_\beta(t)[r] \neq \bot$ and $\mathtt{P}_\beta(t')[r] \neq \bot$. We have $\mathtt{P}_\beta(t)[r] = \mathtt{P}_\beta(t')[r]$.*

Proof. It follows immediately from Lemmas 84 and 50 and Definition 61. □

Lemma 86. *Let β be a correct broker, let σ be a correct server, let r be a root such that $\mathtt{P}[r] \neq \bot$ and $r \in batches$ at σ. Let $b = batches[r]$ at σ, let*

$$\begin{aligned}i &= b.ids \\ p &= b.payloads\end{aligned}$$

We have

$$\begin{aligned}i &= \mathscr{S}(\mathscr{D}(\mathtt{P}[r])) \\ p &= \mathscr{S}(\mathscr{C}(\mathtt{P}[r]))\end{aligned}$$

Proof. It follows immediately from Definition 61, Lemmas 84 and 53, and Lemma 50. □

Lemma 87. *Let β be a correct broker, let r be a root, let $t, t'' \in \mathbb{R}$ such that $t'' > t$, $\mathtt{W}_\beta(t)[r] \neq \bot$ and $\mathtt{W}_\beta(t'')[r] = \bot$. For some $t' \in [t, t'']$ we have $\left|\mathtt{W}_\beta(t')[r]\right| \geq f + 1$.*

Proof. We start by noting that, by Definition 61, $batches[r]$ is `Witnessing` at β at time t. Let t' identify the first moment after t when β updates $batches[r]$'s variant. β does so only by executing line 145, and only if $|batches[r].witnesses| \geq f + 1$ (line 138). We then have $\left|\mathtt{W}_\beta(t')[r]\right| \geq f + 1$, and the lemma is proved. □

Notation 20 (Sequence elements)**.** Let X be a set, let $z \in X^{<\infty}$. We use

$$\{z\} = \{z_n \mid n \leq |z|\} \tag{6.3}$$

to denote the **elements** of z.

Notation 21 (Sequence indexing)**.** Let X be a set, let $z \in X^{<\infty}$ such that all elements of z are distinct. Let $x \in \{z\}$. We use

$$(z_x) = (n \iff z_n = x)$$

to identify the **index** of x in z.

Lemma 88. *Let β be a correct broker, let r be a root such that* $\mathtt{W}[r] \neq \bot$. *β has sent a $\left[\mathtt{Batch}, \tilde{i}, p\right]$ message to all servers, with*

$$\tilde{i} = \mathtt{c}(i)$$
$$i = \mathscr{S}(\mathscr{D}(\mathtt{P}[r]))$$
$$p = \mathscr{S}(\mathscr{C}(\mathtt{P}[r]))$$

Proof. β sets $batches[r]$'s variant to `Witnessing` only by executing line 107. The lemma immediately follows from lines 100, 101, 102, 104 and 105 and Definition 61. □

Lemma 89. *Let β be a correct broker, let r be a root such that* $\mathtt{P}[r] \neq \bot$, *let*

$$\tilde{i} = \mathtt{c}(i)$$
$$i = \mathscr{S}(\mathscr{D}(\mathtt{P}[r]))$$
$$p = \mathscr{S}(\mathscr{C}(\mathtt{P}[r]))$$

Let σ be a correct server. Upon delivering a $\left[\mathtt{Batch}, \tilde{i}, p\right]$ message from β, σ sets

$$batches[r] = Batch\{ids : i, payloads : p, ..\}$$

and sends a $\left[\mathtt{BatchAcquired}, r, u\right]$ message back to β, with $u \subseteq \mathscr{D}(\mathtt{P}[r])$ such that σ knows all elements of $(\mathscr{D}(\mathtt{P}[r]) \setminus u)$.

Proof. Noting that i and p respectively list the domain and codomain of the same map, we have that i has no duplicates and $|i| = \left|p\right|$. Moreover, by Definitions 58 and 61, we have $\mathtt{j}\left(i, p\right) = \mathtt{L}[r]$. By Lemma 84, this proves $r = \rho\left(\mathtt{j}\left(i, p\right)\right)$.

Upon delivering $\left[\mathtt{Batch}, \tilde{i}, p\right]$ (line 64), σ expands $\tilde{i}$ back into $i = \mathtt{e}\left(\tilde{i}\right)$ (line 49) and verifies that i has no duplicates and $|i| = \left|p\right|$ (line 51). σ then computes the set u of ids in $\{i\}$ that σ_n does not know (line 54). We immediately have $u \subseteq \mathscr{D}(\mathtt{P}[r])$, and σ knows all elements of $(\mathscr{D}(\mathtt{P}[r]) \setminus u)$. Next, σ_n computes $r = \rho\left(\mathtt{j}\left(i, p\right)\right)$ (lines 56, 57 and 58). Finally, σ_n sets $batches[r] = Batch\{ids : i, payloads : p, ..\}$ (line 60), and sends a $\left[\mathtt{BatchAcquired}, r, u\right]$ message back to β (lines 61, 67 and 68). □

Lemma 90. *Let β be a correct broker, let r be a root such that, at some point in time, we have* $\mathtt{P}[r] \neq \bot$. *At some point in time we have* $|\mathtt{W}[r]| \geq f + 1$.

Proof. Upon initialization, $batches$ is empty at β (line 47). Moreover, upon adding $(r, _)$ to $batches$ (line 87 only), β sets a $[\mathtt{Reduce}, r]$ timer (line 64). When $[\mathtt{Reduce}, r]$ eventually rings

at β (line 99), β updates $batches[r]$'s variant to `Witnessing`. By Lemma 88, before doing so β sends a $[\texttt{Batch}, \tilde{i}, p]$ message to all servers, with

$$\tilde{i} = \texttt{c}(i)$$
$$i = \mathscr{S}(\mathscr{D}(\texttt{P}[r]))$$
$$p = \mathscr{S}(\mathscr{C}(\texttt{P}[r]))$$

Let $\sigma_1, \ldots, \sigma_{f+1}$ be distinct correct servers (noting that at most f servers are Byzantine, $\sigma_1, \ldots, \sigma_{f+1}$ are guaranteed to exist). Let $n \le f+1$. By Lemma 89, upon delivering $[\texttt{Batch}, \tilde{i}, p]$ σ_n sets $batches[r] = Batch\{ids : i, payloads : p, ..\}$, and sends a $[\texttt{BatchAcquired}, r, u]$ message back to β, with $u \subseteq \mathscr{D}(\texttt{P}[r])$ such that σ knows all elements of $(\mathscr{D}(\texttt{P}[r]) \setminus u)$.

Let us assume that, upon delivering $[\texttt{BatchAcquired}, r, u]$ from σ_n, we have $r \notin batches$ at β. By Definition 61 we have $\texttt{P}[r] = \bot$ which, by Lemma 76, implies $\texttt{W}[r] = \bot$. As a result, by Lemma 87, at some point in time we must have had $|\texttt{W}[r]| \ge f+1$. Throughout the remainder of this proof we assume that, upon delivering $[\texttt{BatchAcquired}, r, u]$ from σ_n, we have $r \in batches$ at β.

Upon delivering $[\texttt{BatchAcquired}, r, u]$ from σ_n (line 116), β verifies that $b \in batches$ (line 117) and, because β knows all elements of u (line 118), β maps u into a corresponding set of assignments a (line 121). Next, β aggregates all elements of $\mathscr{C}(\texttt{Q}[r])$ into a multisignature q (line 123) and copies $\texttt{S}[r]$ into a map s (line 124).

By Lemmas 85 and 81 we have

$$\{i\} \setminus \mathscr{D}(s) = \mathscr{D}(\texttt{P}[r]) \setminus \mathscr{D}(\texttt{S}[r]) = \mathscr{D}(\texttt{Q}[r]) \tag{6.4}$$

By Lemma 82, for all $(\hat{i}, \hat{s}) \in s$, $(\hat{s} = s[\hat{i}] = \texttt{S}[r][i])$ is $\texttt{D}(\hat{i})$'s signature for $[\texttt{Message}, c, m]$, with

$$(c, m) = p_{(i_{\hat{i}})} = \texttt{P}[r][\hat{i}]$$

By Lemma 83 and Equation 6.4, q is $((\{i\} \setminus \mathscr{D}(s)) = \mathscr{D}(\texttt{Q}[r]))$'s multisignature for $[\texttt{Reduction}, r]$.

Having computed a, q and s, β sends a $[\texttt{Signatures}, r, a, q, s]$ message back to σ_n (line 126).

Upon delivering $[\texttt{Signatures}, r, a, q, s]$ from β (line 100), σ_n imports all elements of a (line 72). Noting that a contains assignments for all elements of u, and any element of $(\{i\} \setminus u)$ was known to σ_n upon delivering $[\texttt{Batch}, \ldots]$, σ_n knows all elements of i. Noting that σ_n never modifies or removes elements of $batches$, σ_n successfully retrieves i and p from $batches[r]$ (lines 74 and 77). σ_n then verifies to know all elements of i (line 79).

Next, σ_n loops through all elements of s. For each $(\hat{i}, \hat{s})$ in s, σ_n verifies that $\hat{i} \in i$ (line 83, see Lemma 81), then verifies that $\hat{s}$ is $\mathtt{D}(\hat{i})$'s signature for $[\mathtt{Message}, c, m]$, with $(c, m) = p_{(i_{\hat{i}})}$ (lines 86 and 88). Subsequently, σ_n verifies that q is $(i \setminus \mathscr{D}(s))$'s multisignature for $[\mathtt{Reduction}, r]$ (lines 91 and 93). Finally, σ produces a multisignature w for [`Witness, r`] (line 96) and sends a [`WitnessShard, r, w`] message back to β (lines 97, 103 and 104).

Let us assume that, upon delivering $[\mathtt{WitnessShard}, r, w]$ from σ_n, we have $r \notin batches$ or $batches[r]$ not `Witnessing` at β. By Definition 61 we have $\mathtt{W}[r] = \bot$. As a result, by Lemma 87, at some point in time we must have had $|\mathtt{W}[r]| \geq f+1$. Throughout the remainder of this proof we assume that, upon delivering $[\mathtt{WitnessShard}, r, w]$, we have $r \in batches$ and $batches[r]$ `Witnessing` at β.

Upon delivering $[\mathtt{WitnessShard}, r, w]$ from σ_n (line 129), β verifies that $r \in batches$ and $batches[r]$ is `Witnessing` (lines 130 and 133), then verifies that w is σ_n's multisignature for $[\mathtt{Witness}, r]$ (line 134) and finally adds (σ_n, w) to $batches[r].witnesses$ (line 135).

In summary, for all $n \leq f+1$, either $|\mathtt{W}[r]| \geq f+1$, or β adds a distinct element to $batches[r].witnesses$. Noting that β never removes elements from $batches[r].witnesses$, this trivially reduces to β eventually satisfying $|\mathtt{W}[r]| \geq f+1$, and the lemma is proved. □

Lemma 91. *Let σ be a correct server, let $(r, w) \in witnesses$ at σ. We have that w is a plurality certificate for $[Witness, r]$.*

Proof. Upon initialization, $witnesses$ is empty at σ (line 18). Moreover, σ adds (r, w) to $witnesses$ only by executing line 116. Immediately before doing so, σ verifies that w is a plurality certificate for $[\mathtt{Witness}, r]$ (lines 113 and 114). □

Lemma 92. *Let σ be a correct server, let $t \in \mathbb{R}$, let $(r, b) \in batches$ at σ at time t. For all $t' \geq t$, $(r, b) \in batches$ at σ at time t.*

Proof. Let $t' \geq t$. Because σ never removes elements from $batches$, for some b' we have $(r, b') \in batches$ at σ at time t'. By Lemmas 53 and 50 we have $b' = b$, and the lemma is proved. □

Lemma 93. *Let σ be a correct server, let $((i, c), (_, r)) \in messages$ at σ. We have $r \in batches$ at σ.*

Proof. We start by noting that, upon initialization, $messages$ is empty at σ (line 21). Moreover, σ adds $((i, c), (_, r))$ to $messages$ only by executing line 130. σ does so only if $r \in batches$ (lines 108 and 109). The lemma immediately follows from Lemma 92. □

Lemma 94. *Let σ be a correct server, let $((i,c),(m,r)) \in messages$ at σ. Let $b = batches[r]$ at σ, let*

$$i = b.ids$$
$$p = b.payloads$$

let $l = \mathtt{j}\big(i,p\big)$. We have $(i,c,m) \in \{l\}$.

Proof. We underline that, by Lemma 93, b is well-defined. We start by noting that, upon initialization, $messages$ is empty at σ (line 21). Moreover, σ adds $((i,c),(m,r))$ to $messages$ only by executing line 130. σ does so only if (see Lemma 92) $(i,c,m) \in \{l\}$ (lines 111, 119). □

Lemma 95. *Let β be a correct broker, let r be a root, let $t, t'' \in \mathbb{R}$ such that $t'' > t$, $\mathtt{X}_\beta(t)[r] \neq \bot$ and $\mathtt{X}_\beta\big(t''\big)[r] = \bot$. For some $t' \in \big[t, t''\big]$ we have $\big|\mathtt{X}_\beta\big(t'\big)[r]\big| \geq 2f+1$.*

Proof. We start by noting that, by Definition 61, $batches[r]$ is `Committing` at β at time t. Let t' identify the first moment after t when β updates $batches[r]$'s variant. β does so only by executing line 185, and only if $|batches[r].commits| \geq 2f+1$ (line 171). We then have $\big|\mathtt{X}_\beta\big(t'\big)[r]\big| \geq 2f+1$, and the lemma is proved. □

Lemma 96. *Let β be a correct broker, let r be a root such that $|\mathtt{W}[r]| \geq f+1$. At some point in time we have $|\mathtt{X}[r]| \geq 2f+1$.*

Proof. We start by noting that, if $batches[r]$ is `Witnessing` at broker, β never removes elements from $batches[r].witnesses$, and β updates $batches[r]$'s variant only by executing line 145. As a result, β is eventually guaranteed to detect that $batches[r]$ is `Witnessing` and that $|batches[r].witnesses| \geq f+1$ (line 138). Upon doing so, β aggregates $batch[r].witnesses$ into a certificate c (lines 139 and 140) and sends a [`Witness`, r, c] message to every server (lines 142 and 143).

Let $\sigma_1, \ldots, \sigma_{2f+1}$ be distinct correct servers (noting that at most f servers are Byzantine, $\sigma_1, \ldots, \sigma_{2f+1}$ are guaranteed to exist). Let $n \leq 2f+1$. Noting that $\mathtt{W}[r] \neq \bot$, by Lemma 88, the source-order delivery of perfect links and Lemmas 89 and 92, by the time σ_n handles the delivery of [`Witness`, r, c], σ_n satisfies

$$batches[r] = Batch\big\{ids : i, payloads : p, ..\big\}$$

with

$$i = \mathscr{S}(\mathscr{D}(\mathtt{P}[r]))$$
$$p = \mathscr{S}(\mathscr{C}(\mathtt{P}[r]))$$

Upon delivering $[\texttt{Witness}, r, c]$ (line 136), σ_n verifies that $r \in batches$ (line 108) and retrieves i and p from $batches[r]$ (line 111). σ_n then initializes an empty map $f : \mathbb{I} \rightarrowtail (\mathscr{R}, \mathbb{S}^+, \mathscr{P}, \mathbb{M})$ (line 117), and loops through all elements of $\mathtt{j}(i, p)$ (line 119).

For all $j \leq (|i| = |p|)$, let $(c_j, m_j) = p_j$. Noting that σ_n adds to f only keys that belong to i (line 128), let $k \leq (|i| = |p|)$ such that σ_n sets

$$f[i_k] = (r'_k, w'_k, p'_k, m'_k)$$

By line 120 we immediately have $m'_k \neq m_k$. Additionally, because $w'_k = witnesses[r'_k]$ at σ (line 121), by Lemma 91 w'_k is a plurality certificate for $[\texttt{Witness}, r'_k]$. Finally, by line 120 we have $((i_k, c_k), (m'_k, r'_k)) \in messages$ at σ_n. Let $b'_k = batches[r'_k]$ at σ_n, let

$$l'_k = \mathtt{j}(b'_k.ids, b'_k.payloads)$$

By Lemma 94 we have $(i_k, c_k, m'_k) \in l'_k$. As a result, by lines 122, 124, 126 and 128, we have that p'_k is a proof for (i_k, c_k, m'_k) from r'_k.

Having looped over all elements of $\mathtt{j}(i, p)$, σ_n produces a signature s for $[\texttt{Commit}, r, \mathscr{D}(f)]$ (line 132) and sends a $[\texttt{CommitShard}, r, f, s]$ message back to β (lines 133, 139 and 140).

Let us assume that, upon delivering $[\texttt{CommitShard}, r, f, s]$ from σ_n, we have $r \notin batches$ or $batches[r]$ not $\texttt{Witnessing}$ at β. By Definition 61 we have $\mathtt{X}[r] = \bot$. As a result, by Lemma 95, at some point in time we must have had $|\mathtt{X}[r]| \geq 2f + 1$. Throughout the remainder of this proof we assume that, upon delivering $[\texttt{CommitShard}, r, f, s]$, we have $r \in batches$ and $batches[r]$ $\texttt{Witnessing}$ at β.

Upon delivering $[\texttt{CommitShard}, r, f, s]$ from σ_n (line 148), β verifies that $r \in batches$ and $batches[r]$ is $\texttt{Witnessing}$ (line 149). β then verifies that s is σ_n's multisignature for $[\texttt{Commit}, r, \mathscr{D}(f)]$ (line 150). Recalling that $\mathscr{D}(f) \subseteq \{i\}$ and, by Lemma 85, we still have

$$i = \mathscr{S}(\mathscr{D}(\mathrm{P}[r]))$$
$$p = \mathscr{S}(\mathscr{C}(\mathrm{P}[r]))$$

for all $k \leq |i|$ such that $(i_k, (r'_k, w'_k, p'_k, m'_k)) \in f$ (line 153)), β successfully verifies that: $i_k \in \{i\}$ (line 154); w'_k is a plurality certificate for $[\texttt{Witness}, r'_k]$ (line 159); p'_k is a proof for (i_k, c_k, m'_k) from r'_k (lines 157 and 162); and $m'_k \neq m_k$ (lines 157 and 165). Having done so, β adds $(\sigma_n, _)$ to $batches[r].commits$ (lines 149 and 168).

In summary, for all $n \leq 2f + 1$, either $|\mathtt{X}[r]| \geq 2f + 1$, or β adds a distinct element to $batches[r].commits$. Noting that β never removes elements from $batches[r].commits$, this trivially reduces to β eventually satisfying $|\mathtt{X}[r]| \geq 2f + 1$, and the lemma is proved. □

Lemma 97. *Let β be a correct broker, let r be a root such that* $\mathtt{X}[r] \neq \bot$. *Let* $(\sigma, (i, _)) \in \mathtt{X}[r]$. *We have* $i \subseteq \mathscr{D}(\mathtt{P}[r])$.

Proof. Upon first setting $batches[r]$'s variant to `Committing` (line 145 only), β sets $batches[r].commits = \{\}$. Moreover, β adds $(\sigma, (i, _))$ to $batches[r].commits$ only by executing line 168. Before doing so, β retrieves $b = batches[r]$ (line 149), loops through every element $\hat{i}$ of i (line 153) and verifies $\hat{i} \in b.payloads$ (line 154). The lemma immediately follows from Definition 61. □

Lemma 98. *Let β be a correct broker, let r be a root such that* $\mathtt{P}[r] \neq \bot$, *let* $(i, (c, m)) \in \mathtt{P}[r]$ *such that* $\chi = \mathtt{D}(i)$ *is correct. We have that χ broadcast m for c.*

Proof. Upon initialization, $batches$ is empty at β (line 47). Moreover, β adds (r, b) to $batches$ only by executing line 87. Upon doing so, β sets $b.reductions = \{\}$. By Definition 61 and Lemma 81 we then have $\mathscr{D}(\mathtt{S}[r]) = \mathscr{D}(\mathtt{P}[r])$. Therefore, by Lemma 82, χ signed $[\mathtt{Message}, c, m]$. Because χ does so (line 30 only) only upon broadcasting m for c (line 29), the lemma is proved. □

Lemma 99. *Let σ be a correct server, let r be a root such that σ signs* $[\mathtt{Witness}, r]$. *Some* $l \in (\mathbb{I} \times \mathbb{C} \times \mathbb{M})^{\nearrow}$ *exists such that* $r = \rho(l)$.

Proof. σ signs $[\mathtt{Witness}, r]$ only by executing line 96. σ does so only if $r \in batches$ (lines 74 and 75). Upon initialization, $batches$ is empty at σ (line 17. Moreover, σ adds $(r, _)$ to $batches$ (line 60 only) only if, for some sorted l (lines 49 and 56), we have $r = \rho(l)$ (lines 57 and 58). □

Lemma 100. *Let β be a correct broker, let r be a root such that* $\mathtt{X}[r] \neq \bot$. *Let* $(_, (i, _)) \in \mathtt{X}[r]$, *let* $\hat{i} \in i$, *let* $\chi = \mathtt{D}(\hat{i})$. *We have that χ is Byzantine.*

Proof. We underline that, by Lemmas 97 and 80, χ is well-defined. Upon first setting $batches[r]$'s variant to `Committing` (line 145 only), β sets $batches[r].commits = \{\}$. Moreover, β adds $(_, (i, _))$ to $batches[r].commits$ only by executing line 168. Before doing so, β retrieves $b = batches[r]$ (line 149), then loops through all elements of i (line 153). Upon looping over $\hat{i}$, β retrieves $(c, m) = b.payloads[r]$. β then verifies that, for some root r', at least $f + 1$ servers signed $[\mathtt{Witness}, r']$ (line 159). Next, β verifies, that for some proof p, index n and message m', we have

$$v(r', p, n, (\hat{i}, c, m')) = \mathtt{True} \tag{6.5}$$

(line 162). Finally, β verifies that $m \neq m'$ (line 165).

Let us assume by contradiction that χ is correct. By Lemma 98 we immediately have that χ broadcast m for c. Moreover, noting that at most f servers are Byzantine, at least one correct server signed $[\mathtt{Witness}, r']$. As a result, by Lemma 99, some $l' \in \mathbb{I} \times \mathbb{C} \times \mathbb{M}^{\nearrow}$ exists such that $r' = \rho(l')$. Moreover, by Equation 6.5 and Definition 57, we have $l'_n = (i, c, m')$. By Lemma

68, this proves that χ broadcast m' for c. In summary, χ broadcast m and $m' \neq m$ for c. This contradicts χ being correct and proves the lemma. □

Lemma 101. *Let β be a correct broker, let r be a root such that* $\mathtt{X}[r] \neq \bot$. *Let* $(\sigma, (\epsilon, s)) \in \mathtt{X}[r]$. *We have that s is σ's multisignature for* $[\mathtt{Commit}, r, \epsilon]$.

Proof. Upon setting $batches[r]$'s variant to `Committing` (line 145 only), β initializes $batches[r].commits$ to an empty map. Moreover, β adds $(\sigma, (\epsilon, s))$ to $batches[r].commits$ only by executing line 168. β does so only if s is σ's multisignature for $[\mathtt{Commit}, r, \epsilon]$ (lines 150, 151). □

Lemma 102. *Let β be a correct broker, let r be a root, let $\sigma \in \mathtt{Y}[r]$ be a correct server. We have that $r \in batches$ at server, and σ knows all elements of* $\mathscr{D}(\mathtt{P}[r])$.

Proof. Upon initialization of $batches[r]$, we have $batches[r].commit_to = \emptyset$ at β (line 87 only). Moreover, β adds σ to $batches[r].commit_to$ only by executing line 131. β does so only upon receiving a $[\mathtt{WitnessShard}, r]$ message from σ (line 129). In turn, σ sends a $[\mathtt{WitnessShard}, r]$ message to β only by executing line 104. σ does so only if $r \in batches$ at σ (lines 74 and 75) and β knows all elements of $batches[r].ids$ (lines 88 and 89). By Lemma 86, however, we have $batches[r].ids = \mathscr{S}(\mathscr{D}(\mathtt{P}[r]))$, and the lemma is proved. □

Lemma 103. *Let β be a correct broker, let r be a root such that* $\mathtt{X}[r] \neq \bot$. *We have* $|\mathtt{Y}[r]| \geq f + 1$.

Proof. We start by noting that β updates $batches[r]$'s variant to `Committing` only by executing line 145. β does so only if $batches[r]$ is `Witnessing` and $|batches[r].witnesses| >= f + 1$ (line 138). Upon setting $batches[r]$'s variant to `Witnessing` (line 107 only), β initializes $batches[r].witnesses$ to an empty map. Finally, whenever β adds $(\sigma, _)$ to $batches[r].witnesses$ (line 135 only), β also adds σ to $batches[r].commit_to$ (line 131). The lemma immediately follows from the observation that β never removes elements from $batches[r].commit_to$. □

Lemma 104. *Let β be a correct broker, let r be a root such that* $\mathtt{X}[r] \neq \bot$. *We eventually have* $\mathtt{Z}[r] = \mathtt{True}$.

Proof. We start by noting that β removes r from $batches$ (line 202 only) only if $batches[r]$ is `Completing` (line 194). Moreover, β sets $batches[r]$'s variant to `Completing` (line 185 only) only if $batches[r].committable = \mathtt{True}$. In other words, if $\mathtt{X}[r] \neq \bot$, then β removes r from $batches$ only if $\mathtt{Z}[r] = \mathtt{True}$.

β updates $batches[r]$'s variant to `Committing` (line 145 only) only if $batches[r]$'s variant is `Witnessing`. Immediately after setting $batches[r]$'s variant to `Witnessing` (line 107 only), β sets a $[\mathtt{Committable}, r]$ timer (line 108). When $[\mathtt{Committable}, r]$ eventually rings (line 111), β checks if $r \in batches$ (line 112). If so, β sets $batches[r].committable$ to `True`. Otherwise, as

we proved above, we previously had $batches[r].committable = \texttt{True}$ at β, and the lemma is proved. □

Lemma 105. *Let β be a correct broker, let r be a root such that $|\mathtt{X}[r]| \geq 2f+1$. Let*

$$E = \bigcup_{(_,(\epsilon,_))\in \mathtt{X}[r]} \epsilon$$

Let $\hat{i} \in (\mathscr{D}(\mathtt{P}[r]) \setminus E)$, let $\hat{\chi} = \mathtt{D}(\hat{i})$, let $(\hat{c},_) = \mathtt{P}[r][\hat{i}]$. Some correct server σ exists such that σ eventually delivers a message from $\hat{\chi}$ for $\hat{c}$.

Proof. We start by noting that, if $batches[r]$ is $\texttt{Committing}$ at β, β never removes elements from $batches[r].commits$, and β updates $batches[r]$'s variant only by executing line 185. As a result, by Lemma 104, β is eventually guaranteed to detect that: $batches[r]$ is $\texttt{Committing}$; $batches[r].commitable = true$; and $|batches[r].commits| \geq 2f+1$ (line 171). Upon doing so, β builds a map $p : \mathbb{P}(\mathbb{I}) \rightarrowtail \mathbb{P}(\mathbb{S}^+)$ such that

$$\big((\epsilon,_) \in p\big) \Longleftrightarrow ((_,(\epsilon,_)) \in \mathtt{X}[r]) \tag{6.6}$$

and

$$p[\epsilon] = \bigcup_{(_,(\epsilon,s))\in \mathtt{X}[r]} s \tag{6.7}$$

(lines 174, 176, 177). From p, β builds a map $q : \mathbb{P}(\mathbb{I}) \rightarrowtail \mathbb{S}^+$ that to each ϵ in p associates the aggregation of $p[\epsilon]$ (line 179). By Equation 6.6 we immediately have

$$\bigcup_{(\epsilon,_)\in q} \epsilon = E$$

For all $(_,c) \in q$, let $|c|$ denote the number of signers of c. By Equations 6.6 and 6.7 we have

$$\sum_{(_,c)\in q} |c| = \sum_{(_,s)\in p} |s| = |\mathtt{X}[r]| \geq 2f+1$$

Finally, by Lemma 101, for every $(\epsilon, c) \in q$, c certifies $[\texttt{Commit}, r, \epsilon]$. Having computed q, β sends a $\big[\texttt{Commit}, r, q\big]$ message to all servers in $\mathtt{Y}[r]$ (lines 181 and 182).

By Lemma 103, we have $|\mathtt{Y}[r]| \geq f+1$. Let $\sigma \in \mathtt{X}[r]$ be a correct server. Noting that at most f servers are Byzantine, σ is guaranteed to exist. By Lemma 102, upon receiving a $\big[\texttt{Commit}, r, q\big]$ message from β (line 179), σ verifies that $r \in batches$ (line 144). By Lemma 86, σ then verifies to know all elements $(\{batches[r].ids\} = \mathscr{D}(\mathtt{P}[r]))$ (line 149). Next, σ verifies that, for every $(\epsilon, c) \in q$, c certifies $[\texttt{Commit}, r, \epsilon]$ (lines 154 and 155). Finally, σ verifies that

$$\sum_{(_,c)} |c| \geq 2f+1$$

(lines 152, 154, 158 and 160). Having done so, σ computes E (line 163). Next, by Lemma

86 σ loops through all elements of $\texttt{P}[r]$ (line 166). For every $(i,(c,m)) \in \texttt{P}[r]$ such that $i \notin E$ (line 169), σ either delivers m from $\texttt{D}(i)$ for c (lines 167 and 171), or σ has already delivered a message from $\texttt{D}(i)$ for c (line 169, see Lemma 49). This proves in particular that σ delivers a message from $\hat{\chi}$ for $\hat{c}$, and concludes the lemma. □

Theorem 19. *Draft satisfies validity.*

Proof. Let χ be a correct client, let c be a context, let m be a message such that χ broadcasts m for c. By Lemma 78, either a correct server delivers message from χ for c, or some correct broker β and some root r exist such that eventually $\texttt{P}[r][i] = (c,m)$. Throughout the remainder of this proof, we assume the existence of β and r. By Lemma 90, at some point in time we have $|\texttt{W}[r]| \geq f+1$. Therefore, by Lemma 96, at some point in time we have $|\texttt{X}[r]| \geq 2f+1$. Moreover, by Lemma 100, for all $(_,(\epsilon,_)) \in \texttt{X}[r]$, we have $\texttt{D}(\chi) \notin \epsilon$. As a result, by Lemma 105, some correct server delivers a message from χ for c, and the theorem is proved. □

6.4 Complexity

In this section, we prove to the fullest extent of formal detail the good-case signature and communication complexity of Draft.

6.4.1 Auxiliary results

In this section we gather definitions and lemmas that we will use to prove, in the next sections, the good-case signature and communication complexity of Draft. The results presented in this section hold independently of Draft itself, and could be applicable to a broader spectrum of analyses.

Notation 22 (Bit strings)**.** We use $\mathbb{S} = \{0,1\}^{<\infty}$ to denote all finite strings of bits. We use $\varnothing$ to denote the empty sequence of bits. We use $\mathbb{S}^{<b} = \{0,1\}^{<b}$, $\mathbb{S}^{\leq b} = \{0,1\}^{\leq b}$, $\mathbb{S}^{b} = \{0,1\}^{b}$, $\mathbb{S}^{\geq b} = \{0,1\}^{\geq b}$ and $\mathbb{S}^{>b} = \{0,1\}^{>b}$. We use $\mathbb{S}^{even}$ and $\mathbb{S}^{odd}$ to denote the sets of strings with an even and odd number of bits, respectively. We use programming notation when indexing a string of bits: for all $s \in \mathbb{S}$ we use $s = (s_0, s_1, \ldots)$.

Notation 23 (Cropping)**.** Let $n \in \mathbb{N}$, let $s \in \mathbb{S}$. We use the following **cropping** notation:

$$(s|_{\leq n}) = (s_0, \ldots, s_n)$$
$$(s|_{\geq n}) = \left(s_n, \ldots, s_{|s|-1}\right)$$

We also use $(s|_{<n}) = (s|_{\leq n-1})$ and $(s|_{>n}) = (s|_{\geq n+1})$.

Definition 62 (Integer encoding)**.** For all $b \in \mathbb{N}$, the **b-bits integer representation** $\tilde{\imath}_b$:

$\big(0..(2^b-1)\big) \leftrightarrow \mathbb{S}^b$ is defined by

$$\tilde{\iota}_b(n)_i = \lfloor \frac{n}{2^i} \rfloor \bmod 2$$
$$\tilde{\iota}_b^{-1}(s) = \sum_{i=0}^{b} 2^i s_i$$

The b**-bits integer encoding** $\iota_b : \mathbb{S} \times \big(0..(2^b-1)\big) \leftrightarrow \mathbb{S}^{\geq b}$ is defined by

$$\iota_b(s,n) = \tilde{\iota}_b(n) \frown s$$
$$\iota_b^{-1}(s) = \big((s|_{\geq b}), \tilde{\iota}_b^{-1}((s|_{<b}))\big)$$

Lemma 106. *Let* $b \in \mathbb{N}$*.* $\tilde{\iota}_b^{-1}$ *is injective.*

Proof. Let $s, s' \in \mathbb{S}$ such that $s \neq s'$. Let

$$k = \max i \mid s_i \neq s'_i$$

Noting that $s \neq s'$, k is guaranteed to exist. Without loss of generality, let us assume $s_k = 1$, $s'_k = 0$. We by Definition 62 have

$$\begin{aligned}\tilde{\iota}_b^{-1}(s) - \tilde{\iota}_b^{-1}(s') &= \sum_{i=0} 2^i (s_i - s'_i) = \\ &= \sum_{i=0}^{k-1} 2^i (s_i - s'_i) + 2^k \geq \\ &\geq \sum_{i=0}^{k-1} -2^i + 2^k \geq 1\end{aligned}$$

which proves $\tilde{\iota}_b^{-1}(s) \neq \tilde{\iota}_b^{-1}(s')$ and concludes the lemma □

Lemma 107. *Let* $b \in \mathbb{N}$*, let* $s \in \mathbb{S}^b$*. We have* $\tilde{\iota}_b\big(\tilde{\iota}_b^{-1}(s)\big) = s$*.*

Proof. By Definition 62 we have

$$\begin{aligned}
\tilde{\iota}_b\big(\tilde{\iota}_b^{-1}(s)\big)_i &= \lfloor \frac{\sum_{j=0}^{b} 2^j s_j}{2^i} \rfloor \bmod 2 = \\
&= \lfloor \frac{\sum_{j=0}^{i-1} 2^j s_j}{2^i} + s_i + \frac{\sum_{j=i+1}^{b} 2^j s_j}{2^i} \rfloor \bmod 2 = \\
&= \lfloor \underbrace{\frac{\sum_{j=0}^{i-1} 2^j s_j}{2^i}}_{<1} + \underbrace{s_i + 2\sum_{j=0}^{b-i-1} 2^j s_{j+i+1}}_{\in \mathbb{N}} \rfloor \bmod 2 = \\
&= \left(s_i + 2\underbrace{\sum_{j=0}^{b-i-1} 2^j s_{j+i+1}}_{\in \mathbb{N}} \right) \bmod 2 = s_i
\end{aligned}$$

□

Lemma 108. *Let $b \in \mathbb{N}$. $\tilde{\iota}_b$ is is a bijection.*

Proof. It follows immediately from Lemmas 106 and 107. □

Lemma 109. *Let $b \in \mathbb{N}$. ι_b is a bijection.*

Proof. It follows immediately from Definition 62 and Lemma 108. □

Definition 63 (Varint encoding)**.** The **varint representation** $\tilde{\nu} : \mathbb{N}^+ \leftrightarrow \mathbb{S}^{even}$ is defined by

$$\begin{aligned}
|\tilde{\nu}(n)| &= 2\lceil \log_2(n+1) \rceil \\
\tilde{\nu}(n)_i &= \begin{cases} 1 & \text{iff } i \bmod 2 = 0, i < 2\lceil \log_2(n+1)\rceil - 2 \\ 0 & \text{iff } i = 2\lceil \log_2(n+1)\rceil - 2 \\ \tilde{\iota}_{\lceil \log_2(n+1)\rceil}(n)_{\frac{i-1}{2}} & \text{otherwise} \end{cases} \\
\tilde{\nu}^{-1}(s) &= \tilde{\iota}^{-1}_{\frac{|s|}{2}}\big(s_1, s_3, s_5, \ldots, s_{|s|-1}\big)
\end{aligned}$$

The set of **varint-parseable strings** is the set

$$\mathbb{S}^{\nu} = \{ s \in \mathbb{S} \mid (\exists k \in \mathbb{N} \mid s_{2k} = 0) \}$$

The **varint encoding** $\nu : \big(\mathbb{S} \times \mathbb{N}^+ \leftrightarrow \mathbb{S}^{\nu}\big)$ is defined by

$$\begin{aligned}
\nu(s, n) &= \tilde{\nu}(n) \frown s \\
\nu^{-1}(s) &= \Big(\big(s|_{\geq \lambda(s)}\big), \tilde{\nu}^{-1}\big(\big(s|_{<\lambda(s)}\big)\big)\Big)
\end{aligned}$$

with $\lambda : \mathbb{S} \to \mathbb{N}$ defined by

$$\lambda(s) = 2(\min k \mid s_{2k} = 0) + 2$$

Definition 63 contains a slight abuse of notation: we prove below that $\tilde{\nu}^{-1}$ inverts $\tilde{\nu}$, but we do not prove that $\tilde{\nu}$ inverts $\tilde{\nu}^{-1}$ (it does not). Similarly we prove that ν^{-1} inverts ν, but not that ν inverts ν^{-1}.

Lemma 110. *Let $n \in \mathbb{N}^+$. We have $\tilde{\nu}^{-1}(\tilde{\nu}(n)) = n$.*

Proof. By Definition 63 and Lemma 109 we have

$$\begin{aligned}\tilde{\nu}^{-1}(\tilde{\nu}(n)) &= \tilde{\iota}^{-1}_{\frac{2\lceil\log_2(n+1)\rceil}{2}}\left(\tilde{\iota}_{\lceil\log_2(n+1)\rceil}(n)_0, \tilde{\iota}_{\lceil\log_2(n+1)\rceil}(n)_1, \ldots, \tilde{\iota}_{\lceil\log_2(n+1)\rceil}(n)_{\lceil\log_2(n+1)\rceil-1}\right) = \\ &= \tilde{\iota}^{-1}_{\log_2(n+1)}\left(\tilde{\iota}_{\log_2(n+1)}(n)\right) = n\end{aligned}$$

□

Lemma 111. *Let let $s \in \mathbb{S}$, let $n \in \mathbb{N}^+$. We have $\nu^{-1}(\nu(s,n)) = (s,n)$.*

Proof. All derivations in this lemma follow directly from Definition 63. We have

$$\nu(s,n) = \tilde{\nu}(n) \frown s$$

Moreover, we have

$$\forall k < \lceil\log_2 n+1\rceil - 1, \nu(s,n)_{2k} = \tilde{\nu}(n)_{2k} = 1$$

and

$$\nu(s,n)_{2(\lceil\log_2(n+1)\rceil-1)} = \tilde{\nu}^{-1}(n)_{2(\lceil\log_2(n+1)\rceil-1)} = 0$$

which proves

$$\lambda(\nu(s,n)) = 2\lceil\log_2(n+1)\rceil$$

This implies

$$\left(\nu(s,n)|_{<\lambda(\nu(s,n))}\right) = \left(\tilde{\nu}(n) \frown s|_{<2\lceil\log_2(n+1)\rceil}\right) = \tilde{\nu}(n)$$

and similarly

$$\left(\nu(s,n)|_{\geq\lambda(\nu(s,n))}\right) = s$$

By Lemma 110 we then have

$$\nu^{-1}(\nu(s,n)) = \left(s, \tilde{\nu}^{-1}(\tilde{\nu}(n))\right) = (s,n)$$

and the lemma is proved. □

Notation 24 (Integer partition)**.** Let X be a finite set, let $\mu : X \to \mathbb{P}^{<\infty}(\mathbb{N})$. We call μ a **integer partition** on X. We use

$$\begin{aligned} |\mu| &= \sum_{x \in X} |\mu(x)| \\ \max \mu &= \max_{x \in X} \max \mu(x) \end{aligned}$$

Notation 25 (Enumeration)**.** Let $X = \{x_1, \ldots, x_{|X|}\}$ be a finite set. We use $\mathscr{E}(X) = (x_1, \ldots, x_{|X|})$ to denote any specific **enumeration** of X.

Definition 64 (Partition representation)**.** Let X be a finite set. The **partition representation** on X is the function $\rho : (X \to \mathbb{P}^{<\infty}(\mathbb{N})) \leftrightarrow \mathbb{S}$ defined, with

$$(x_1, \ldots, x_{|X|}) = \mathscr{E}(X)$$

by

$$\begin{aligned} w(\mu) &= \lceil \log_2 (\max \mu + 1) \rceil \\ d(\mu) &= \lceil \log_2 (|\mu| + 1) \rceil \\ y_n(\mu) &= |\mu(x_n)| \\ \left(z_1(\mu), \ldots, z_{|\mu|}(\mu)\right) &= \mathscr{E}(\mu(x_1)) \frown \ldots \frown \mathscr{E}(\mu(x_{|X|})) \\ \mathtt{v}_0(\mu) &= \varnothing \\ \mathtt{v}_n(\mu) &= \iota_{w(\mu)}(\mathtt{v}_{n-1}(\mu), z_n(\mu)) \\ \mathtt{v}(\mu) &= \mathtt{v}_{|\mu|}(\mu) \\ \mathtt{k}_0(\mu) &= \mathtt{v}(\mu) \\ \mathtt{k}_n(\mu) &= \iota_{d(\mu)}(\mathtt{k}_{n-1}(\mu), y_n(\mu)) \\ \mathtt{k}(\mu) &= \mathtt{k}_{|X|}(\mu) \\ \mathtt{g}(\mu) &= \nu(\mathtt{k}(\mu), w(\mu)) \\ \mathtt{h}(\mu) &= \nu(\mathtt{g}(\mu), d(\mu)) \\ \rho(\mu) &= \mathtt{h}(\mu) \end{aligned}$$

and

$$
\begin{aligned}
\mathtt{h}(s) &= s \\
\big(\mathtt{g}(s), d(s)\big) &= \nu^{-1}(\mathtt{h}(s)) \\
(\mathtt{k}(s), w(s)) &= \nu^{-1}\big(\mathtt{g}(s)\big) \\
\mathtt{k}_{|X|}(s) &= \mathtt{k}(s) \\
\big(\mathtt{k}_{n-1}(s), y_n(s)\big) &= \iota^{-1}_{d(s)}(\mathtt{k}_n(s)) \\
t(s) &= \sum_{n=1}^{|X|} y_n(s) \\
\mathtt{v}(s) &= \mathtt{k}_0(s) \\
\mathtt{v}_{t(s)}(s) &= \mathtt{v}(s) \\
(\mathtt{v}_{n-1}(s), z_n(s)) &= \iota^{-1}_{w(s)}(\mathtt{v}_n(s)) \\
p_n(s) &= \sum_{i=1}^{n} y_n(s) \\
\rho^{-1}(s)(x_n) &= \big\{z_{p_{n-1}(s)+1}(s), \dots, z_{p_n(s)}(s)\big\}
\end{aligned}
$$

Definition 64 contains a slight abuse of notation: we prove below that ρ^{-1} inverts ρ, but ρ is obviously not defined on all of $\mathbb{S}$ ($\varnothing$ being a trivial counterexample).

Lemma 112. *Let X be a finite set, let $\mu : X \to \mathbb{P}^{<\infty}(\mathbb{N})$. We have $\rho^{-1}\big(\rho(\mu)\big) = \mu$.*

Proof. All derivations in this lemma follow from Notation 24, Definition 64 and Lemmas 109 and 111. Let $s = \rho(\mu)$. We trivially have

$$\mathtt{h}(s) = s = \rho(\mu) = \mathtt{h}(\mu)$$

In steps, we can derive

$$\big(\mathtt{g}(s), d(s)\big) = \nu^{-1}(\mathtt{h}(s)) = \nu^{-1}\big(\mathtt{h}(\mu)\big) = \nu^{-1}\big(\nu\big(\mathtt{g}(\mu), d(\mu)\big)\big) = \big(\mathtt{g}(\mu), d(\mu)\big)$$

$$(\mathtt{k}(s), w(s)) = \nu^{-1}\big(\mathtt{g}(s)\big) = \nu^{-1}\big(\mathtt{g}(\mu)\big) = \nu^{-1}\big(\nu\big(\mathtt{k}(\mu), w(\mu)\big)\big) = \big(\mathtt{k}(\mu), w(\mu)\big)$$

and

$$\mathtt{k}_{|X|}(s) = \mathtt{k}(s) = \mathtt{k}(\mu) = \mathtt{k}_{|X|}(\mu)$$

By backwards induction, for all $n \in 1..|X|$ we then have

$$
\begin{aligned}
\big(\mathtt{k}_{n-1}(s), y_n(s)\big) &= \iota^{-1}_{d(s)}(\mathtt{k}_n(s)) = \iota^{-1}_{d(\mu)}\big(\mathtt{k}_n(\mu)\big) = \\
&= \iota^{-1}_{d(\mu)}\Big(\iota_{d(\mu)}\big(\mathtt{k}_{n-1}(\mu), y_n(\mu)\big)\Big) = \big(\mathtt{k}_{n-1}(\mu), y_n(\mu)\big)
\end{aligned}
$$

from which follows

$$t(s) = \sum_{n=1}^{|X|} y_n(s) = \sum_{n=1}^{|X|} y_n(\mu) = \sum_{n=1}^{|X|} |\mu(x_n)| = |\mu|$$

In steps, we can then derive

$$\mathtt{v}(s) = \mathtt{k}_0(s) = \mathtt{k}_0(\mu) = \mathtt{v}(\mu)$$

$$\mathtt{v}_{|\mu|}(s) = \mathtt{v}_{t(s)}(s) = \mathtt{v}(s) = \mathtt{v}(\mu) = \mathtt{v}_{|\mu|}(\mu)$$

Again by backwards induction, for all $n \in 1..|\mu|$ we then have

$$\begin{aligned}(\mathtt{v}_{n-1}(s), z_n(s)) &= \iota_{w(s)}^{-1}(\mathtt{v}_n(s)) = \iota_{w(\mu)}^{-1}(\mathtt{v}_n(\mu)) \\ &= \iota_{w(\mu)}^{-1}\left(\iota_{w(\mu)}(\mathtt{v}_{n-1}(\mu), z_n(\mu))\right) = (\mathtt{v}_{n-1}(\mu), z_n(\mu))\end{aligned}$$

In summary we have

$$\begin{aligned}&\forall n \in 1..|X|, y_n(s) = y_n(\mu) \\ &\forall n \in 1..|\mu|, z_n(s) = z_n(\mu)\end{aligned}$$

from which finally follows

$$\begin{aligned}\rho^{-1}(s)(x_n) &= \{z_{p_{n-1}(s)+1}(s), \ldots, z_{p_n(s)}(s)\} = \\ &= \{z_{p_{n-1}(s)+1}(\mu), \ldots, z_{p_n(s)}(\mu)\} = \\ &= \left\{z_{(\sum_{i=1}^{n-1} y_i(s)+1)}(\mu), \ldots, z_{(\sum_{i=1}^{n} y_i(s))}(\mu)\right\} = \\ &= \left\{z_{(\sum_{i=1}^{n-1} y_i(\mu)+1)}(\mu), \ldots, z_{(\sum_{i=1}^{n} y_i(\mu))}(\mu)\right\} = \\ &= \left\{z_{(\sum_{i=1}^{n-1} |\mu(x_i)|+1)}(\mu), \ldots, z_{(\sum_{i=1}^{n} |\mu(x_i)|)}(\mu)\right\} = \\ &= \left\{z_{(\sum_{i=1}^{n-1} |\mathscr{E}(\mu(x_i))|+1)}(\mu), \ldots, z_{(\sum_{i=1}^{n} |\mathscr{E}(\mu(x_i))|)}(\mu)\right\} = \\ &= \left\{\mathscr{E}(\mu(x_n))_1, \ldots, \mathscr{E}(\mu(x_n))_{|\mu(x_n)|}\right\} = \mu(x_n)\end{aligned}$$

which proves $\rho^{-1}(\rho(\mu)) = \rho^{-1}(s) = \mu$, and concludes the lemma. □

Lemma 113. *Let X be a finite set, let $\mu : \mathbb{N} \to \left(X \to \mathbb{P}^{<\infty}(\mathbb{N})\right)$ be a sequence of partition represen-ations such that*

$$\begin{aligned}&\lim_{n\to\infty} |\mu_n| = \infty \\ &\lim_{n\to\infty} \frac{\log_2(\log_2(\max \mu_n))}{|\mu_n|} = 0\end{aligned}$$

We have

$$\lim_{n\to\infty} \frac{|\rho(\mu_n)|}{|\mu_n|} = \lceil \log_2(\max \mu_n + 1) \rceil$$

Proof. All derivations in this proof follow from Definitions 64, 63 and 62. Let $n \in \mathbb{N}$. We have

$$\left|\mathtt{v}_0(\mu_n)\right| = 0$$

and, for all $k \in 1..\left|\mu\right|$,

$$\left|\mathtt{v}_k(\mu_n)\right| = \left|\mathtt{v}_{k-1}(\mu_n)\right| + w(\mu_n) = \left|\mathtt{v}_{k-1}(\mu_n)\right| + \lceil \log_2 (\max \mu_n + 1) \rceil$$

from which, by induction, follows

$$\left|\mathtt{v}(\mu_n)\right| = \left|\mu_n\right| \lceil \log_2 (\max \mu_n + 1) \rceil$$

Similarly we have

$$\left|\mathtt{k}_0(\mu_n)\right| = \left|\mu_n\right| \lceil \log_2 (\max \mu + 1) \rceil$$

and, for all $k \in 1..|X|$,

$$\left|\mathtt{k}_k(\mu_n)\right| = \left|\mathtt{k}_{k-1}(\mu_n)\right| + d(\mu_n) = \left|\mathtt{k}_{k-1}(\mu_n)\right| + \lceil \log_2 (\left|\mu_n\right| + 1) \rceil$$

from which, by induction, follows

$$\left|\mathtt{k}(\mu_n)\right| = \left|\mu_n\right| \lceil \log_2 (\max \mu_n + 1) \rceil + |X| \lceil \log_2 (\left|\mu_n\right| + 1) \rceil$$

In steps we can then derive

$$\begin{aligned} &\left|\mathtt{g}(\mu_n)\right| = \left|\mathtt{k}(\mu_n)\right| + 2 \lceil \log_2 (w(\mu_n) + 1) \rceil = \\ &= \left|\mu_n\right| \lceil \log_2 (\max \mu_n + 1) \rceil + |X| \lceil \log_2 (\left|\mu_n\right| + 1) \rceil + 2 \lceil \log_2 (\lceil \log_2 (\max \mu_n + 1) \rceil + 1) \rceil \end{aligned}$$

and finally

$$\begin{aligned} &\left|\rho(\mu_n)\right| = \left|\mathtt{h}(\mu_n)\right| = \left|\mathtt{g}(\mu_n)\right| + 2 \lceil \log_2 (d(\mu_n)) \rceil = \\ &= \left|\mu_n\right| \lceil \log_2 (\max \mu_n + 1) \rceil + |X| \lceil \log_2 (\left|\mu_n\right| + 1) \rceil + 2 \lceil \log_2 (\lceil \log_2 (\max \mu_n + 1) \rceil + 1) \rceil \\ &\quad + 2 \lceil \log_2 (\lceil \log_2 (\left|\mu_n\right| + 1) \rceil + 1) \rceil \end{aligned}$$

The above holds for any $n \in \mathbb{N}$. Moreover, by hypothesis we have

$$\lim_{n \to \infty} \frac{|X| \lceil \log_2 (\left|\mu_n\right| + 1) \rceil}{\left|\mu_n\right|} = 0$$
$$\lim_{n \to \infty} \frac{2 \lceil \log_2 (\lceil \log_2 (\left|\mu_n\right| + 1) \rceil + 1) \rceil}{\left|\mu_n\right|} = 0$$

and

$$\lim_{n\to\infty} \frac{2\lceil\log_2\left(\lceil\log_2\left(\max\mu_n+1\right)\rceil+1\right)\rceil}{|\mu_n|} =$$

$$\lim_{n\to\infty} \frac{2\lceil\log_2\left(\lceil\log_2\left(\max\mu_n\right)\rceil\right)\rceil}{|\mu_n|} \frac{\lceil\log_2\left(\lceil\log_2\left(\max\mu_n+1\right)\rceil+1\right)\rceil}{\lceil\log_2\left(\lceil\log_2\left(\max\mu_n\right)\rceil\right)\rceil} = 0$$

This proves that

$$\lim_{n\to\infty} \frac{|\rho(\mu_n)|}{|\mu_n|} = \lceil\log_2\left(\max\mu_n+1\right)\rceil$$

and concludes the lemma. □

6.4.2 Batching limit

As we discussed in Section 5.1, Draft is designed to asymptotically match, in the good case, the signature and communication complexity of Oracle-CSB, a toy implementation of CSB that relies on an infallible oracle to uphold all CSB properties. As we discussed, Oracle-CSB achieves optimal signature complexity (as it requires no signature verification) and optimal communication complexity (if the frequency at which clients broadcast their payloads is uniform or unknown). Below we establish and discuss Draft's **batching limit**, i.e., the collection of assumptions and limits at which Draft matches Oracle-CSB's complexity:

- **Assumption: good-case execution**. As we discussed in Section 5.3.1, in the good case: *links are synchronous* (messages are delivered at most one time unit after they are sent); *all processes are correct*; and *the set of brokers contains only one element.* **Discussion.** We assume only one broker for pedagogical reasons: as long as every broker is exposed to a high enough rate of submissions, all derivations in this section still hold true. In the real world, the assumptions of synchrony and correctness are not strict for clients: if a small fraction of broadcasting clients is slow or Byzantine, Draft still achieves near-oracular efficiency, linearly degrading its performance as more client fail to engage with the broker to reduce its batch[2].

- **Assumption: steady-state directory**. We assume that *all servers know all broadcasting clients.* **Discussion.** This assumption is naturally satisfied if all broadcasting clients have already broadcast at least one message. In a real-world, long-lived system, most clients can safely be assumed to broadcast more than one message (this is especially true in the context of a cryptocurrency). Similarly to client synchrony and correctness, Draft's performance degrades linearly in the number of unknown, broadcasting clients.

- **Assumption: concurrent broadcasts**. We assume that *all clients broadcast their message within b time units of each other*, where b is Draft's batching window parameter (see Sections 6.2.3 and 6.2.4). **Discussion.** Similarly to broker count, this assumption is made

[2] Proving this result is beyond the scope of this Part, and left as an exercise to the interested reader.

for pedagogical reasons: as long as the rate at which payloads are submitted to each broker is high enough, all derivations in this section still hold true.

- **Limit: infinite broadcasts**. We derive Draft's complexity at the limit of *infinitely many broadcasting clients*. **Discussion.** All limits derived in this section converge approximately inversely with the number of broadcasting clients. This means that even a finite, real-world implementation of Draft can achieve near-oracular efficiency.

- **Limit: sub-double-exponential clients**. We assume that *the number of clients is infinitely small with respect to the exponential of the exponential of the number of broadcasting clients*. **Discussion.** This limit is only technical, and trivially satisfied by any realistic number of broadcasting clients.

6.4.3 Protocol analysis

In this section, we establish Draft's signature and communication complexity at the batching limit.

Theorem 20. *At the batching limit, a Draft server delivers a payload p by performing 0 signature verifications and exchanging at most $\left(\lceil \log_2(c) \rceil + |p|\right)$ bits.*

Proof. All derivations in this proof follow from the batching limit's assumptions and limits (see Section 6.4.2). Let β denote the only broker in the system. Let $\chi_1, \ldots, \chi_M$ denote the set of broadcasting clients. We have $M \to \infty$. For all j, let $i_j = \mathtt{D}(\chi_j)$, let $p_j = (c_j, m_j)$ identify the payload broadcast by χ_j, let a_j be i_j's id assignment certificate. Without loss of generality we assume that for all $j < j'$ we have $i_j < i_{j'}$. The goal of this proof is to compute, for all $j \in 1..M$, p_j's maximum amortized signature complexity ψ_j and p_j's maximum amortized bit complexity χ_j.

Let $n \in 1..M$. Without loss of generality, we assume that χ_n triggers $\langle cl.\text{Broadcast}\rangle$ (line 29) between time 0 and b. Upon doing so, χ produces a signature s_n for $[\mathtt{Message}, c_n, m_n]$ (line 30), then sends a $[\mathtt{Submission}, a_n, (c_n, m_n, s_n)]$ message to β (line 40).

β receives $[\mathtt{Submission}, a_n, (c_n, m_n, s_n)]$ (line 50) between time 0 and time $B+1$: indeed, χ_n's `Submission` message takes between 0 and 1 time units to reach β. Upon delivering χ_n's `Submission` message, β pushes

$$y_n = Submission\{context: c_n, message: m_n, signature: s_n\}$$

to $pending[i_n]$ (line 54), detects that $pending[i_n]$ is not empty (line 57), empties $pending[i_n]$ (line 58) and adds (i_n, y_n) to $pool$ (line 59). Initially, we have $collecting =$ `False` and $pool = \emptyset$ at β. Moreover, β adds its first element to $pool$ no earlier than time 0. Upon doing so (line 62), β sets a [`Flush`] timer to ring at some time $t_f \geq B+1$ (line 64).

In summary, for all $j \in 1..M$, β adds (i_j, y_j) to $pool$ by time $B+1$, and [Flush] rings at time $t_f \geq B+1$. When [Flush] rings (line 67), β takes $(i_1, p_1), \ldots, (i_M, p_M)$ from $pool$ (line 70) and computes $r = \rho(\mathrm{j}(i, p))$ (line 76). For each $j \in 1..M$, β produces an inclusion proof q_j for (i_j, c_j, m_j) in r (line 80), then sends an $[\texttt{Inclusion}, c_j, r, q_j]$ message to χ_j. Next, β sets

$$batches[r] = Reducing\{payloads: p, signatures: s, \ldots\}$$

(line 87). Finally, β sets a $[\texttt{Reduce}, r]$ to ring at some time $t_r \geq t_f + 2$ (line 89).

χ_n receives $[\texttt{Inclusion}, c_n, r, q_n]$ by time $t_f + 1$ (line 50). Upon doing so, χ_n produces a multi-signature g_n for $[\texttt{Reduction}, r]$ (line 54), then sends a $[\texttt{Reduction}, r, g_n]$ message to β (line 55).

β receives $[\texttt{Reduction}, r, g_n]$ by time t_r (line 92). Because $[\texttt{Reduce}, r]$ has not yet rung, $batches[r]$ is still Reducing at β (line 93). β adds (i_n, g_n) to $batches[r].reductions$ (line 95), then removes (i_n, s_n) from $batches[r].signatures$ (line 96).

In summary, before $[\texttt{Reduce}, r]$ rings at time t_r, for all $j \in 1..M$, β added (i_j, g_j) to $batches[r].reductions$ and removed (i_j, s_j) from $batches[r].signatures$. This means in particular that, when $[\texttt{Reduce}, r]$, $batches[r].signatures$ is empty at β. Upon ringing $[\texttt{Reduce}, r]$ (line 99), β builds the integer partition $\tilde{i}: \mathbb{D} \rightarrowtail \mathbb{P}^{<\infty}(\mathbb{N})$ defined by

$$\left((d, k) \in \tilde{i}\right) \Longleftrightarrow ((d, k) \in i)$$

(line 101). We have $\left|\tilde{i}\right| = |i| = M$. Moreover, by the density of Directory, we have $\max \tilde{i} \leq c - 1$. Recalling that $M \rightarrow \infty$ and $\log(\log(c))/M \rightarrow \infty$, by Lemmas 112 and 113, $\tilde{i}$ can be represented by a string of bits $\hat{i}$ such that

$$\left|\hat{i}\right| = M\lceil \log_2(c) \rceil + o(M)$$

To each server, β sends a

$$x^{(b)} = [\texttt{Batch}, \tilde{i}, p]$$

message (line 105). We have

$$\begin{aligned}\left|x^{(b)}\right| &= M\lceil \log_2(c) \rceil + \sum_{j=1}^{M} \left|p_j\right| + o(M) = \\ &= \sum_{j=1}^{M} \left(\lceil \log_2(c) \rceil + \left|p_j\right| + o(1)\right)\end{aligned}$$

Consequently, for all $j \in 1..M$, $x^{(b)}$'s amortized size for p_j is

$$\left|x^{(b)}\right|_j = \left(\lceil \log_2(c) \rceil + \left|p_j\right|\right)$$

Having sent $x^{(b)}$ to all servers, β updates $batches[r]$ to Witnessing (line 107).

Let σ be a server. Upon receiving $\left[\texttt{Batch}, \tilde{i}, p\right]$ (line 64), σ observes to know all ids in $i_1, \ldots, i_M$ (line 54), computes r (line 58) then sends back to β a

$$x^{(ba)} = \left[\texttt{BatchAcquired}, r, \emptyset\right]$$

message (line 68). Noting that

$$\left|x^{(ba)}\right| = O(1) = \sum_{j=1}^{M} o(1)$$

for all $j \in 1..M$, $x^{(ba)}$'s amortized size for p_j is

$$\left|x^{(ba)}\right|_j = 0$$

Upon receiving σ's $\left[\texttt{BatchAcquired}, r, \emptyset\right]$ (line 116), β exports no assignment (line 121), aggregates all $g_1, \ldots, g_M$ into a single g (line 123), observes $batches[r].signatures$ to be empty, then sends back to σ a

$$x^{(s)} = \left[\texttt{Signatures}, r, \emptyset, g, \emptyset\right]$$

message (line 126). Noting that $\left|x^{(s)}\right| = O(1)$, $x^{(s)}$'s amortized size for p_j is

$$\left|x^{(s)}\right|_j = 0$$

Upon receiving $\left[\texttt{Signatures}, r, \emptyset, g, \emptyset\right]$ (line 100), σ verifies only g (line 91, the loop at line 82 does no iterations). Noting that σ performs signature verifications only upon receiving a `Signatures` message, and noting that

$$1 = \sum_{1}^{M} o(1)$$

we can immediately derive

$$\psi_j = 0$$

Next, σ produces a multi-signature w_σ for $[\texttt{Witness}, r]$ (line 96), then sends back to β a

$$x_\sigma^{(ws)} = [\texttt{WitnessShard}, r, w_\sigma]$$

message (line 104). Noting that $\left|x_\sigma^{(ws)}\right| = O(1)$, $x_\sigma^{(ws)}$'s amortized size for p_j is

$$\left|x_\sigma^{(ws)}\right|_j = 0$$

Upon receiving $f+1$ `WitnessShard` messages (lines 129, 135, 138), β aggregates all witnesses

into a certificate w (line 140), then sends a

$$x^{(w)} = [\texttt{Witness}, r, w]$$

message to all servers (line 143). Noting that $\left|x^{(w)}\right| = O(1)$, $x^{(w)}$'s amortized size for p_j is

$$\left|x^{(w)}\right|_j = 0$$

Upon receiving $[\texttt{Witness}, r, w]$ (line 136), σ observes that, for all j, p_j is not equivocated (lines 119 and 120) (because all clients are correct, no client equivocates), produces a multi-signature c_σ for $[\texttt{Commit}, r, \emptyset]$ (line 132), then sends a

$$x_\sigma^{(cs)} = [\texttt{CommitShard}, r, \emptyset, c_\sigma]$$

message (line 140). Noting that $\left|x_\sigma^{(cs)}\right| = O(1)$, $x_\sigma^{(cs)}$'s amortized size for p_j is

$$\left|x_\sigma^{(cs)}\right|_j = 0$$

Upon receiving $2f+1$ CommitShard messages (lines 148, 168 and 171), β builds a map $h : \mathbb{P}(\mathbb{I}) \rightarrowtail \mathbb{S}^+$ with only the key $\emptyset$ (lines 176, 177, 179) (we recall that every server produced an empty exception set for r). β then sends a

$$x^{(c)} = [\texttt{Commit}, r, h]$$

message to all servers (line 182). Noting that $|x^{(c)}| = O(1)$, $x^{(c)}$'s amortized size for p_j is

$$\left|x^{(c)}\right|_j = 0$$

Upon delivering $[\texttt{Commit}, r, h]$ (line 179), σ delivers $p_1, \ldots, p_M$ (line 171), produces a multi-signature z_σ for $\left[\texttt{Completion}, r, \emptyset\right]$ (line 175), then sends a

$$x_\sigma^{(zs)} = \left[\texttt{CompletionShard}, r, z_\sigma\right]$$

message to β (line 183). Noting that $\left|x_\sigma^{(zs)}\right| = O(1)$, $x_\sigma^{(zs)}$'s amortized size for p_j is

$$\left|x_\sigma^{(zs)}\right|_j = 0$$

Finally, σ sets an $\left[\texttt{OfferTotality}\right]$ timer to ring after 7 time units (173). Summarizing the scheduling of messages we have that: every server delivers the Batch message between time t_r and $t_r + 1$; β delivers all BatchAcquired messages between time t_r and $t_r + 2$; every server delivers a Signatures message between time t_r and $t_r + 3$; β delivers all WitnessShard messages between time t_r and $t_r + 4$; every server delivers the Witness message between

time t_r and $t_r + 5$; β delivers all `CommitShard` messages between time t_r and $t_r + 6$; all servers deliver the `Commit` message between time t_r and $t_r + 7$. As a result, $[\texttt{OfferTotality}, r]$ rings at σ after all servers delivered $p_1, \ldots, p_M$. Upon ringing $[\texttt{OfferTotality}, r]$ (line 186), σ sends a

$$x^{(ot)} = [\texttt{OfferTotality}, r]$$

message to all servers (line 188).

Let σ' be a server. Upon receiving σ's $[\texttt{OfferTotality}, r]$ message, σ' verifies to have already delivered a batch with root r and no exclusions (line 192) and ignores the message.

In summary, noting that σ exchanges $2n$ `OfferTotality` messages (N outgoing, N incoming), σ's amortized bit complexity for p_j is

$$\psi_j^\sigma = \left|x^{(b)}\right|_j + \left|x^{(ba)}\right|_j + \left|x^{(s)}\right|_j + \left|x_\sigma^{(ws)}\right|_j + \left|x^{(w)}\right|_j + \left|x_\sigma^{(cs)}\right|_j + \left|x^{(c)}\right|_j + \left|x_\sigma^{(zs)}\right|_j + 2n\left|x^{(ot)}\right|_j$$

from which immediately follows

$$\psi_j^\sigma = \left(\lceil \log_2 (c) \rceil + |p_j|\right)$$

and

$$\psi_j = \max_\sigma \psi_j^\sigma = \left(\lceil \log_2 (c) \rceil + |p_j|\right)$$

which concludes the theorem. □

7 Dibs

In this chapter, we present in detail the **Directory** abstraction and discuss its properties. We then present Dibs, an algorithm that implements Directory, and prove its **security**.

7.1 Interface

Notation 26 (Signatures and multi-signatures)**.** We use $\mathbb{S}^1$ and $\mathbb{S}^+$ to respectively denote the set of signatures and multisignatures.

Definition 65 (Id)**.** An **id** is an element of $(\mathbb{I} = \mathbb{D} \times \mathbb{N})$, where $\mathbb{D}$ is a finite set of domains. Let $i = (d, n)$ be an id, we call d and n the **domain** and **index** of i, respectively.

The **Directory** interface (instance dir) exposes the following procedures and events:

- **Request** $\langle dir.\text{Signup}\rangle$: requests that an id be assigned to the local process.
- **Indication** $\langle dir.\text{SignupComplete}\rangle$: indicates that an id was successfully assigned to the local process.
- **Getter** $dir[id]$: returns the process associated with id id, if known. Otherwise, returns $\perp$.
- **Getter** $dir[process]$: returns the id associated with process $process$, if known. Otherwise, returns $\perp$.
- **Getter** $dir.export(id)$: returns the assignment for id id, if known. Otherwise, returns $\perp$.
- **Setter** $dir.import(assignment)$: imports assignment $assignment$.

Notation 27 (Bijective relations)**.** Let X, Y be sets. We use $(X \leftrightarrow Y)$ to denote the set of **bijective relations** between X and Y.

Notation 28 (Tuple binding)**.** Let $t = (t_1, \ldots, t_n)$ be a tuple. When binding t, we use the **any** symbol _ to mark which elements of t are discarded from the binding. For example,

$$\left(x, _, \ldots, _, y, _\right) = t$$

binds $x = t_1$, and $y = t_{n-1}$. We use the **tail** symbol .. to indicate that all subsequent elements of t are discarded from the binding. For example,

$$\left(x', y', ..\right) = t$$

binds $x' = t_1$ and $y' = t_2$, regardless of n. Let X be a set. We use the tuple binding notation to **filter** the elements X. For example, we use $\left\{\left(x'', _, y'', ..\right)\right\}$ to identify the set

$$\left\{t \in X \mid |t| \geq 3, \left(x'', _, y'', ..\right) = t\right\}$$

of tuples in X whose first element is x'' and whose third element is y''.

Definition 66 (Directory record)**.** Let $\mathtt{D} : (\Pi_C \times \mathbb{R}) \to (\mathbb{I} \leftrightarrow \Pi)$. $\mathtt{D}$ is a **directory record** if and only if:

- For all $\pi \in \Pi_C$, $\mathtt{D}_\pi(t)$ is non-decreasing in t.
- For all $\pi, \pi' \in \Pi_C$, $t, t' \in \mathbb{R}$, $\mathtt{D}_\pi(t) \cup \mathtt{D}_{\pi'}\left(t'\right)$ is a a bijective relation.

Let π be a correct process, let t be a time, let i be an id, let ρ be a process. π **associates** i and ρ by time t if and only if $\left(i, \rho\right) \in \mathtt{D}_\pi(t)$. π **knows** i (resp., ρ) by time t if and only if $(i, _) \in \mathtt{D}_\pi(t)$ (resp., $\left(_, \rho\right) \in \mathtt{D}_\pi(t)$).

Notation 29 (Directory record)**.** Let π be a correct process. Wherever it can be unequivocally inferred from context, we omit the time from the directory record $\mathtt{D}_\pi$.

A Directory satisfies the following properties:

- **Correctness**: Some directory record $\mathtt{D}$ exists such that, for any id i and process ρ, if a process π invokes $dir[i]$ (resp., $dir[\rho]$), π obtains ρ (resp., i) if and only if π associates i and ρ.
- **Signup Integrity**: A correct process never triggers $\langle dir.\text{SignupComplete}\rangle$ before triggering $\langle dir.\text{Signup}\rangle$.
- **Signup Validity**: If a correct process triggers $\langle dir.\text{Signup}\rangle$, it eventually triggers $\langle dir.\text{SignupComplete}\rangle$.
- **Self-knowledge**: Upon triggering $\langle dir.\text{SignupComplete}\rangle$, a correct process knows itself.

- **Transferability**: If a correct process invokes $dir.export(i)$ to obtain an assignment a, then any correct process knows i upon invoking $dir.import(a)$.

- **Density**: For every correct process π, for every time t, for every $((_, n), _) \in \mathtt{D}_\pi(t)$, we have $n < |\Pi|$.

Notation 30 (Directory mapping)**.** Let i be an id, let ρ be a process such that, for some $\pi \in \Pi_C$, $t \in \mathbb{R}$, we have $(i, \rho) \in \mathtt{D}_\pi(t)$. We say that i and ρ are **known**, and we use

$$i = \mathtt{D}(\rho) \quad \rho = \mathtt{D}(i)$$

We underline the soundness of Notation 30: by Definition 66, if $(i, \rho) \in \mathtt{D}_\pi(t)$, then no $i' \neq i$ or $\rho' \neq \rho$ exist such that, for some π', t', we have $(i, \rho') \in \mathtt{D}_{\pi'}(t')$ or $(i', \rho) \in \mathtt{D}_{\pi'}(t')$. In other words, no $i' \neq i$ or $\rho' \neq \rho$ exist such that $i' = \mathtt{D}(\rho)$ or $\rho' = \mathtt{D}(i)$.

7.2 Algorithm

In this section, we present Dibs's pseudocode. The remainder of Chapter 7 proves Dibs's security to the fullest extent of formal detail.

7.2.1 Pseudocode (Client)

```
1  implements:
2      Directory, instance dir
3
4
5  uses:
6      AuthenticatedPointToPointLinks, instance al
7
8
9  struct Assignment:
10     id: Id,
11     process: Process,
12     certificate: Certificate
13
14
15 enum Status(Outsider, SigningUp, SignedUp)
16
17
18 upon <dir.Init>:
19     rankings: {Server: {Server}} (default {}) = {};
20     assigner: (Server or ⊥) = ⊥;
```

```
21      assignments: {Integer: {Server: MultiSignature}} (default {}) =
      {};
22      status: Status = Outsider;
23
24      directory: {(Id, Process)} = {};
25      certificates: {Id: Certificate} = {};
26
27
28  upon <dir.Signup>:
29      status = SigningUp;
30
31      for σ in Σ:
32          trigger <al.Send | σ, [Signup]>;
33
34
35  upon <al.Deliver | σ, [Ranked, source]>:
36      rankings[source].add(σ);
37
38
39  upon exists source such that |rankings[source]| >= f + 1 and assigner
      = ⊥:
40      assigner = source;
41
42      for σ in Σ:
43          trigger <al.Send | σ, [Assigner, assigner]>;
44
45
46  upon event <al.Deliver | σ, [Assignment, index, signature]>:
47      if σ.multiverify(signature, [Assignment, (assigner, index), self])
      :
48          assignments[index][σ] = signature;
49
50
51  upon exists index such that |assignments[index]| >= 2f+1 and status =
      SigningUp:
52      status = SignedUp;
53
54      id = (assigner, index);
55      certificate = aggregate(assignments[index]);
56
57      dir.import(Assignment {id, process: self, certificate});
58      trigger <dir.SignupComplete>;
```

```
59
60
61 procedure dir.[] (query):
62     if query is Id:
63         if exists process such that (query, process) in directory:
64             return process;
65     else if query is Process:
66         if exists id such that (id, query) in directory:
67             return id;
68
69     return ⊥;
70
71
72 procedure dir.export(id):
73     if let certificate = certificates[id]:
74         process = dir[id];
75         return Assignment {id, process, certificate};
76     else:
77         return ⊥;
78
79
80 procedure dir.import(assignment):
81     if assignment.certificate.verify_quorum(
           [Assignment, assignment.id, assignment.process]
       ):
82         directory.add((assignment.id, assignment.process));
83         certificates[assignment.id] = assignment.certificate;
```

7.2.2 Pseudocode (Server)

```
1  implements:
2      DirectoryServer, instance dsr
3
4
5  uses:
6      FifoBroadcast, instance fb
7      AuthenticatedPointToPointLinks, instance al
8
9
10 upon <dsr.Init>:
11     rankings: {Server: [KeyCard]} (default []) = {};
12     assigners: {KeyCard: Server} = {};
```

```
13      certified: {KeyCard} = {};
14
15
16  upon <al.Deliver | π, [Signup]>:
17      trigger <fb.Broadcast | [Rank, π]>;
18
19
20  upon <fb.Deliver | σ, [Rank, process]>:
21      if process not in rankings[σ]:
22          rankings[σ].push_back(process);
23          trigger <al.Send | process, [Ranked, σ]>
24
25
26  upon <al.Deliver | π, [Assigner, assigner]>:
27      if π not in assigners:
28          assigners[π] = assigner;
29
30
31  upon exists process such that (process in assigners)
        and (process in rankings[assigners[process]])
        and (process not in certified):
32      certified.add(process);
33
34      assigner = assigners[process];
35      index = rankings[assigner].index_of(process);
36
37      signature = multisign([Assignment, (assigner, index), process]);
38      trigger <al.Send | process, [Assignment, index, signature]>;
```

7.3 Correctness

In this section, we prove to the fullest extent of formal detail that Dibs implements a Directory.

7.3.1 Correctness

In this section, we prove that Dibs satisfies correctness.

Lemma 114. *Let σ be a correct server, let $\hat{\sigma}$ be a server. All elements of $ranking[\hat{\sigma}]$ at σ are distinct.*

Proof. Upon initialization, $rankings[\hat{\sigma}]$ is empty at σ (line 11). Moreover, σ appends π to

$rankings[\hat{\sigma}]$ only by executing line 22. Because σ does so only if $\pi \notin rankings[\hat{\sigma}]$ (line 21), the lemma is proved. □

Lemma 115. *Let σ, σ' be correct servers, let $\hat{\sigma}$ be a server, let $n \in \mathbb{N}$, let π be a process. If $rankings[\hat{\sigma}][n] = \pi$ at σ, then eventually $rankings[\hat{\sigma}][n] = \pi$ at σ' as well.*

Proof. Let $[\texttt{Rank}, \pi_1], \ldots, [\texttt{Rank}, \pi_R]$ denote the sequence of $[\texttt{Rank}, _]$ messages σ FIFO-delivered from $\hat{\sigma}$. By the totality and FIFO properties of FIFO broadcast, σ' eventually delivers $[\texttt{Rank}, \pi_1], \ldots, [\texttt{Rank}, \pi_R]$ as well. Upon initialization, $rankings[\hat{\sigma}]$ is empty at both σ and σ' (line 11). Moreover, σ (resp., σ') adds some process $\hat{\pi}$ to $rankings[\hat{\sigma}]$ (line 22 only) only upon FIFO-delivering a $[\texttt{Rank}, \hat{\pi}]$ message from $\hat{\sigma}$ (line 20), and only if $rankings[\hat{\sigma}]$ satisfies a deterministic condition on $\hat{\pi}$ (line 21). As a result, noting σ (resp., σ') never removes elements from $rankings[\hat{\sigma}]$, if $rankings[\hat{\sigma}][n] = \pi$ at σ as a result of σ delivering $[\texttt{Rank}, \pi_1], \ldots, [\texttt{Rank}, \pi_R]$, then eventually $rankings[\hat{\sigma}][n] = \pi$ at σ' as well, as it also eventually delivers $[\texttt{Rank}, \pi_1], \ldots, [\texttt{Rank}, \pi_R]$. □

Lemma 116. *Let σ, σ' be correct servers. Let $\hat{\sigma}$ be a server, let $n \in \mathbb{N}$, let π, π' be processes such that σ and σ' respectively sign $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi\right]$ and $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi'\right]$. We have $\pi = \pi'$.*

Proof. σ (resp., σ') signs $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi\right]$ (resp., $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi'\right]$) only by executing line 37. σ (resp., σ') does so only if n is the index of π (resp., π') in $rankings[\hat{\sigma}]$ at σ (resp., σ') (line 35). We underline that, by Lemma 114, at most one element of $rankings[\hat{\sigma}]$ is π (resp., π') at σ (resp., σ'), hence n is well-defined. By Lemma 115, however, if $rankings[\hat{\sigma}][n] = \pi$ at σ, then $rankings[\hat{\sigma}][n] = \pi$ at σ' as well. This proves $\pi = \pi'$ and concludes the lemma. □

Lemma 117. *Let $\hat{\sigma}$ be a server, let $n \in \mathbb{N}$, let π, $\pi' \neq \pi$ be processes. If a quorum certificate for $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi\right]$ exists, then no quorum certificate for $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi'\right]$ exists.*

Proof. Let us assume by contradiction that quorum certificates exist for both $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi\right]$ and $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi'\right]$. Noting that at most f servers are Byzantine, at least one correct server σ (resp., σ') signed $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi\right]$ (resp., $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi'\right]$). By Lemma 116 we then have $\pi = \pi'$, which contradicts $\pi \neq \pi'$ and proves the lemma. □

Lemma 118. *Let σ, σ' be correct servers. Let $\hat{\sigma}$ be a server, let $n, n' \in \mathbb{N}$, let π be a process such that σ and σ' respectively sign $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi\right]$ and $\left[\texttt{Assignment}, \left(\hat{\sigma}, n'\right), \pi\right]$, We have $n = n'$.*

Proof. σ (resp., σ') signs $\left[\texttt{Assignment}, (\hat{\sigma}, n), \pi\right]$ (resp., $\left[\texttt{Assignment}, \left(\hat{\sigma}, n'\right), \pi\right]$) only by executing line 37. σ (resp., σ') does so only if n (resp., n') is the index of π in $rankings[\hat{\sigma}]$ at σ (resp., σ') (line 35). We underline that, by Lemma 114, at most one element of

$rankings[\hat{\sigma}]$ is π at σ (resp., σ'), hence n (resp., n') is well-defined. By Lemma 115, however, if $rankings[\hat{\sigma}][n] = \pi$ at σ, then $rankings[\hat{\sigma}][n] = \pi$ at σ' as well. Again by Lemma 114, this proves $n = n'$ and concludes the lemma. □

Lemma 119. *Let σ be a correct server. Let $\hat{\sigma}$, $\hat{\sigma}'$ be servers, let π be a process such that σ signs both an $[\mathit{Assignment}, (\hat{\sigma}, _), \pi]$ and an $[\mathit{Assignment}, (\hat{\sigma}', _), \pi]$ message. We have $\hat{\sigma} = \hat{\sigma}'$.*

Proof. σ signs an $[\texttt{Assignment}, (\hat{\sigma}, _), \pi]$ (resp., $[\texttt{Assignment}, (\hat{\sigma}', _), \pi]$) message only by executing 37. σ does so only if $\hat{\sigma} = assigners[\pi]$ (resp., $\hat{\sigma}' = assigners[\pi]$) (line 34). The lemma follows immediately from the observation that σ sets $assigners[\pi]$ (line 28 only) at most once (line 27). □

Lemma 120. *Let $\hat{\sigma}$, $\hat{\sigma}'$ be servers, let $n, n' \in \mathbb{N}$ such that $(\hat{\sigma}, n) \neq (\hat{\sigma}', n')$, let π be a process. If a quorum certificate for $[\mathit{Assignment}, (\hat{\sigma}, n), \pi]$ exists, then no quorum certificate for $[\mathit{Assignment}, (\hat{\sigma}', n'), \pi]$ exists.*

Proof. Let us assume that $\hat{\sigma} = \hat{\sigma}'$. Because $(\hat{\sigma}, n) \neq (\hat{\sigma}', n')$, we have $n \neq n'$. Let us assume by contradiction that quorum certificates exist for both $[\texttt{Assignment}, (\hat{\sigma}, n), \pi]$ and $[\texttt{Assignment}, ((\hat{\sigma} = \hat{\sigma}'), n'), \pi]$. Noting that at most f servers are Byzantine, at least one correct server σ (resp. σ') signed an $[\texttt{Assignment}, (\hat{\sigma}, n), \pi]$ (resp., $[\texttt{Assignment}, (\hat{\sigma}, n'), \pi]$) message. By Lemma 118 we then have $n = n'$, which contradicts $n \neq n'$.

Let us assume that $\hat{\sigma} \neq \hat{\sigma}'$. We have that at least $f + 1$ correct processes signed $[\texttt{Assignment}, (\hat{\sigma}, _), \pi]$ (resp., $[\texttt{Assignment}, (\hat{\sigma}', _), \pi]$). Because any two sets of $f + 1$ correct processes intersect in at least one element, some correct server σ^* exists that signed both $[\texttt{Assignment}, (\hat{\sigma}, _), \pi]$ and $[\texttt{Assignment}, (\hat{\sigma}', _), \pi]$. By Lemma 119 we then have $\hat{\sigma} = \hat{\sigma}'$ which contradicts $\hat{\sigma} \neq \hat{\sigma}'$ and proves the lemma. □

Definition 67 (Certifiable assignments)**.** Let i be an id, let π be a process. The set $\mathtt{D}^{\infty}$ of **certifiable assignments** contains (i, π) if and only if a quorum certificate ever exists for $[\texttt{Assignment}, i, \pi]$.

Lemma 121. *$\mathtt{D}^{\infty}$ is a bijection.*

Proof. It follows immediately from Definition 67 and Lemmas 117 and 120. □

Notation 31 (Directory record)**.** Let π be a correct process, let $t \in \mathbb{R}$. We use $\mathtt{D}_\pi(t)$ to denote the value of $directory$ at π at time t.

As we immediately prove, $\mathtt{D}$ is a directory record, which makes Notation 31 compatible with Definition 66.

Lemma 122. *Let π be a correct process, let $t, t' \in \mathbb{R}$ such that $t' \geq t$. We have $\mathtt{D}_\pi(t') \supseteq \mathtt{D}_\pi(t)$.*

Proof. The lemma immediately follows from Notation 31 and the observation that π never removes elements from $directory$. □

Lemma 123. *Let π be a correct process, let $t \in \mathbb{R}$. We have $\mathtt{D}_\pi(t) \subseteq \mathtt{D}^\infty$.*

Proof. Upon initialization, $directory$ is empty at π (line 24). Moreover, π adds $(\hat{\imath}, \hat{\pi})$ only by executing line 82. π does so only upon verifying a quorum certificate for $[\mathtt{Assignment}, \hat{\imath}, \hat{\pi}]$ (line 81). The lemma immediately follows from Definition 67 and Notation 31. □

Lemma 124. $\mathtt{D}$ *is a directory record.*

Proof. By Lemma 122 we immediately have that, for every correct process π, $\mathtt{D}_\pi(t)$ is non-decreasing in t. Let π, π' be correct processes, let $t, t' \in \mathbb{R}$. By Lemma 123 we have

$$\mathtt{D}_\pi(t) \cup \mathtt{D}_{\pi'}(t') \subseteq \mathtt{D}^\infty$$

By Lemma 121, this proves that $\mathtt{D}_\pi(t) \cup \mathtt{D}_{\pi'}(t')$ is bijective. The lemma immediately follows from Definition 66. □

Theorem 21. *Dibs satisfies correctness.*

Proof. Let π be a correct process, let $\hat{\imath}$ be an id, let $\hat{\pi}$ be a process. By Notation 31, upon invoking $dir[\hat{\imath}]$ (resp., $dir[\hat{\pi}]$) (line 61), π obtains $\hat{\pi}$ (resp., $\hat{\imath}$) (line 64, resp., line 67) if and only if $(\hat{\imath}, \hat{\pi}) \in \mathtt{D}_\pi$ (line 63, resp., line 66). The theorem follows immediately from Lemma 124. □

7.3.2 Signup integrity

In this section, we prove that Dibs satisfies signup integrity.

Theorem 22. *Dibs satisfies signup integrity.*

Proof. Let π be a correct process. Upon initialization, we have $status = \mathtt{Outsider}$ at π (line 22). Moreover, π triggers $\langle dir.SignupComplete \rangle$ only by executing line 58. π does so only if $status = \mathtt{SigningUp}$ (line 51). The theorem follows immediately from the observation that π sets $status = \mathtt{SigningUp}$ (line 29 only) only upon triggering $\langle dir.Signup \rangle$ (line 28). □

7.3.3 Signup validity

In this section, we prove that Dibs satisfies signup validity.

Lemma 125. *Let π be a correct process that triggers $\langle dir.Signup \rangle$. We eventually have $assigner \neq \bot$ at π.*

Proof. Upon triggering $\langle dir.Signup \rangle$ (line 28), π sends a $[\texttt{Signup}]$ message to all servers (lines 31 and 32). Let $\hat{\sigma}$ be a correct server. Noting that at most f servers are Byzantine, $\hat{\sigma}$ is guaranteed to exist.

Upon delivering $[\texttt{Signup}]$ from π (line 16), $\hat{\sigma}$ FIFO-broadcasts a $[\texttt{Rank}, \pi]$ message (line 17). Let $\sigma_1, \ldots, \sigma_{f+1}$ be distinct correct servers. Noting that at most f servers are Byzantine, $\sigma_1, \ldots, \sigma_{f+1}$ are guaranteed to exist.

Let $n \leq f+1$. We start by noting that, upon initialization, $rankings[\hat{\sigma}]$ is empty at σ_n (line 11). Moreover, σ_n adds π to $rankings[\hat{\sigma}]$ (line 22) only upon delivering a $[\texttt{Rank}, \pi]$ message from $\hat{\sigma}$ (line 20). By the validity and totality of FIFO broadcast, σ_n is eventually guaranteed to FIFO-deliver $[\texttt{Rank}, \pi]$ from $\hat{\sigma}$. Upon doing so (line 20), σ_n verifies that $\pi \notin rankings[\hat{\sigma}]$ (line 22), then sends a $[\texttt{Ranked}, \hat{\sigma}]$ message to π.

Upon delivering a $[\texttt{Ranked}, \hat{\sigma}]$ message from σ_n (line 35), π adds σ_n to $rankings[\hat{\sigma}]$ (line 36). In summary, recalling that the above holds true for any $n \leq f+1$, π adds $\sigma_1, \ldots, \sigma_{f+1}$ to $rankings[\hat{\sigma}]$. Noting that π never removes elements from $rankings[\hat{\sigma}]$, that initially $assigner = \bot$ at π (line 20), and that π updates $assigner$ only by executing line 40, π is eventually guaranteed to detect that, for some σ^*, $\left|rankings[\sigma^*]\right| \geq f+1$ and $assigner = \bot$ (line 39). Upon doing so, π sets $assigner = \sigma^*$, and the lemma is proved. □

Lemma 126. *Let π be a correct process, let $\hat{\sigma}$ be a server such that $assigner = \hat{\sigma}$ at π. We eventually have $\pi \in rankings[\hat{\sigma}]$ at all correct processes.*

Proof. We start by noting that, upon initialization, we have $assigner = \bot$ at π (20). Moreover, π sets $assigner = \hat{\sigma}$ (line 40 only) only if $\left|rankings[\hat{\sigma}]\right| \geq f+1$ at π (line 39). Upon initialization, $rankings[\hat{\sigma}]$ is empty at π (line 19). Moreover, π adds σ to $rankings[\hat{\sigma}]$ only upon delivering a $[\texttt{Ranked}, \hat{\sigma}]$ message from σ. In summary, at least $f+1$ servers sent a $[\texttt{Ranked}, \hat{\sigma}]$ message to π. Let σ be a correct server that sent a $[\texttt{Ranked}, \hat{\sigma}]$ message to π. Noting that at most f servers are Byzantine, σ is guaranteed to exist.

σ sends a $[\texttt{Ranked}, \hat{\sigma}]$ to π (line 23 only) only upon FIFO-delivering a $[\texttt{Rank}, \pi]$ message from $\hat{\sigma}$. By the totality of FIFO broadcast, eventually every correct process delivers $[\texttt{Rank}, \pi]$. Let σ' be a correct process. Upon FIFO-delivering $[\texttt{Rank}, \pi]$ from $\hat{\sigma}$ (line 20), σ' either already satisfies $\pi \in rankings[\hat{\sigma}]$ (line 21) or adds π to $rankings[\hat{\sigma}]$. Noting that no correct process ever removes elements from $rankings[\hat{\sigma}]$, eventually $\pi \in rankings[\hat{\sigma}]$ at all correct processes, and the lemma is proved. □

Lemma 127. *Let π be a correct process, let $\hat{\sigma}$ be a server such that $assigner = \hat{\sigma}$ at π, let $r \in \mathbb{N}$ such that, eventually, $rankings[\hat{\sigma}][r] = \pi$ at all correct servers. We eventually have $\left|assignments[r]\right| \geq 2f+1$ at π.*

Proof. We start by noting that, upon initialization, we have $assigner = \bot$ at π (line 20). Moreover, π updates $assigner$ to $\hat{\sigma}$ only by executing line 40. Upon doing so, π sends a $[\texttt{Assigner}, \hat{\sigma}]$ to all servers (lines 42 and 43). Moreover, because π updates $assigner$ to a non-$\bot$ value (line 40) immediately before disseminating $[\texttt{Assigner}, \hat{\sigma}]$, and π never resets $assigner$ back to $\bot$, π never issues any $[\texttt{Assigner}, (\tilde{\sigma} \neq \hat{\sigma})]$ message (see line 39). Let $\sigma_1, \ldots, \sigma_{2f+1}$ be distinct correct servers. Noting that at most f servers are Byzantine, $\sigma_1, \ldots, \sigma_{2f+1}$ are guaranteed to exist.

Let $n \leq 2f+1$. Upon initialization, $assigners$ is empty at σ_n (line 12). Moreover, σ_n adds π to $assigners$ (line 28 only) only upon delivering a $[\texttt{Assigner}, _]$ message from π (line 26). As a result, upon delivering $[\texttt{Assigner}, \hat{\sigma}]$ from π (line 26), σ_n verifies $\pi \notin assigners$ (line 27) and sets $assigners[\pi] = \hat{\sigma}$ (line 28). Upon initialization, $certified$ is empty at σ_n (line 13). Moreover, σ_n adds π to $certified$ only by executing line 32. As a result, σ_n is eventually guaranteed to observe $assigners[\pi] = \hat{\sigma}$, $rankings[\hat{\sigma}] = r$, and $\pi \notin certified$ (line 31). Upon doing so, σ_n produces a signature s_n for $[\texttt{Assignment}, (\hat{\sigma}, r), \pi]$ (lines 34, 35 and 37) and sends an $[\texttt{Assignment}, r, s_n]$ message back to π (line 38).

Upon delivering $[\texttt{Assignment}, r, s_n]$ from σ_n (line 46), π verifies s_n against $[\texttt{Assignment}, (\hat{\sigma}, r), \pi]$ (line 47), then adds σ_n to $assignments[r]$. In summary, recalling that the above holds true for all $n \leq 2f+1$, $assignments[r]$ eventually contains $\sigma_1, \ldots, \sigma_{2f+1}$ at π, and the lemma is proved.

□

Theorem 23. *Dibs satisfies signup validity.*

Proof. Let π be a correct process that triggers $\langle dir.\text{Signup}\rangle$. By Lemma 125, for some $\hat{\sigma}$ we eventually have $assigner = \hat{\sigma}$ at π. By Lemmas 126 and 115, some $r \in \mathbb{N}$ exists such that, eventually, $rankings[\hat{\sigma}][r] = \pi$ at all correct processes. By Lemma 127, we eventually have $|assignments[r]| \geq 2f+1$ at π. Upon triggering $\langle dir.Signup\rangle$ (line 28), π sets $status = \texttt{SigningUp}$ (line 29). Moreover, if $status = \texttt{SigningUp}$, π updates $status$ only by executing line 52. As a result, π is eventually guaranteed to detect that $|assignments[r]| \geq 2f+1$ and $status = \texttt{SigningUp}$ (line 51). Upon doing so, π triggers $\langle dir.\text{SignupComplete}\rangle$ (line 58), and the theorem is proved. □

7.3.4 Self-knowledge

In this section, we prove that Dibs satisfies self-knowledge.

Lemma 128. *Let π be a correct process. Let $\hat{\sigma}$ be a server such that $assigner = \hat{\sigma}$ at π. Let $r \in \mathbb{N}$, let σ be a server such that, for some s, we have $assignments[r][\sigma] = s$ at π. We have that s is σ's signature for $[Assignment, (\hat{\sigma}, r), \pi]$.*

Proof. We start by noting that π updates $assigner$ (line 40 only) only if $assigner = \bot$ (line 46). Consequently, at all times we either have $assigner = \bot$ or $assigner = \hat{\sigma}$ at π. Upon initialization, $assignments$ is empty at π (line 21). Moreover, π sets $assignments[r][\sigma] = s$ only by executing line 48. π does so only if s is σ's multisignature for $[\texttt{Assignment}, (\hat{\sigma}, r), \pi]$ (line 47). □

Theorem 24. *Dibs satisfies self-knowledge.*

Proof. Let π be a correct process that triggers $\langle dir.SignupComplete \rangle$. π does so (line 58 only) only if, for some r, we have $|assignments[r]| \geq 2f + 1$ at π (line 51). Let $\hat{\sigma}$ denote the value of $assigner$ at π, let $i = (\hat{\sigma}, r)$. Immediately before triggering $\langle dir.\text{SignupComplete} \rangle$, π aggregates $assignments[r]$ into a certificate t (line 55). By Lemma 128, t is a quorum certificate for $[\texttt{Assignment}, i, \pi]$. π then invokes

$$dir.import\big(Assignment\{id : i, process : \pi, certificate : t\}\big)$$

(line 57). Upon doing so, π verifies that t is indeed a certificate for $[\texttt{Assignment}, i, \pi]$ (line 81), and adds (i, π) to $directory$ (line 82). By Notation 31 and Definition 66, π knows itself, and the theorem is proved. □

7.3.5 Transferability

In this section, we prove that Dibs satisfies transferability.

Lemma 129. *Let π be a correct process, let i be an id, be a process such that $(i, _) \in directory$ at π. We have $i \in certificates$ at π.*

Proof. Upon initialization, $directory$ is empty at π (line 24). Moreover, π adds $(i, _)$ to $directory$ only by executing line 82. Immediately after doing so, π adds i to $certificates$ (line 83). □

Lemma 130. *Let π be a correct process, let $(\hat{i}, \hat{\pi}) \in directory$ at π. Let $\hat{c} = certificates[\hat{i}]$ at π. We have that $\hat{c}$ is a quorum certificate for $[\texttt{Assignment}, \hat{i}, \hat{\pi}]$.*

Proof. We underline that, by Lemma 129, $\hat{c}$ is guaranteed to exist. By Notation 31 and Lemmas 123 and 121, for all $\hat{\pi}' \neq \hat{\pi}$ we always have $(\hat{i}, \hat{\pi}') \notin directory$ at π. Upon initialization, $certificates$ is empty at π (line 25). Moreover, π adds $(\hat{i}, \hat{c})$ to $certificates$ only by executing line 83. π does so only if $\hat{c}$ is a quorum certificate for $[\texttt{Assignment}, \hat{i}, \tilde{\pi}]$ for some $\tilde{\pi}$ (line 81). Immediately before adding $(\hat{i}, \hat{c})$ to $certificates$, however, π adds $(\hat{i}, \tilde{\pi})$ to $directory$ (line 82). This proves that $\tilde{\pi} = \hat{\pi}$ and concludes the lemma. □

Theorem 25. *Dibs satisfies transferability.*

Proof. Let π, π' be correct processes, let $\hat{\imath}$ be an id such that $a = dir.export(\hat{\imath})$ at π is an assignment. Let

$$\tilde{\imath} = a.id$$
$$\hat{\pi} = a.process$$
$$\hat{c} = a.certificate$$

By lines 75 and 72, we immediately have $\tilde{\imath} = \hat{\imath}$. Moreover, by line 73 we have $\hat{c} = certificates[\hat{\imath}]$ at π. Finally, by line 74 we have $(\hat{\imath}, \hat{\pi}) \in directory$ at π. By Lemma 130, we then have that $\hat{c}$ is a quorum certificate for $[\texttt{Assignment}, \hat{\imath}, \hat{\pi}]$.

Upon invoking $dir.import(a)$, π' verifies that $\hat{c}$ is indeed a quorum certificate for $[\texttt{Assignment}, \hat{\imath}, \hat{\pi}]$ (line 81), then adds $(i, _)$ to $directory$ (line 82). By Notation 31 and Definition 66, π' knows $\hat{\imath}$, and the theorem is proved. □

7.3.6 Density

In this section, we prove that Dibs satisfies density.

Lemma 131. *Let σ be a correct server, let $\hat{\sigma}$ be a server. We have $\left|rankings[\hat{\sigma}]\right| < |\Pi|$ at σ.*

Proof. Upon initialization, $rankings[\hat{\sigma}]$ is empty at σ (line 19). Moreover, σ adds a process π to $rankings[\hat{\sigma}]$ (line 22 only) only if π is not already in $rankings[\hat{\sigma}]$ (line 21). This proves that, at σ, the elements of $rankings[\hat{\sigma}]$ are distinct processes. We then have $\left|rankings[\hat{\sigma}]\right| < |\Pi|$ at σ, and the lemma is concluded. □

Lemma 132. *Let (d, n) be an id, let π be a process. If a quorum certificate exists for $[Assignment, (d, n), \pi]$, then $n < |\Pi|$.*

Proof. Let us assume that a quorum certificate exists for $[\texttt{Assignment}, (d, n), \pi]$. Noting that at most f servers are Byzantine, at least one correct server σ signed $[\texttt{Assignment}, (d, n), \pi]$. σ does so (line 37 only) only if $n < \left|rankings[\hat{\sigma}]\right|$, for some server $\hat{\sigma}$ (line 35). By Lemma 131 we then have $n < |\Pi|$, and the lemma is proved. □

Theorem 26. *Dibs satisfies density.*

Proof. Let π be a correct process, let $t \in \mathbb{R}$, let $((\hat{d}, \hat{n}), \hat{\pi}) \in \mathtt{D}_\pi(t)$. By Lemma 123 we have $((\hat{d}, \hat{n}), \hat{\pi}) \in \mathtt{D}^\infty$. By Definition 67 and Lemma 132 we then have $n < |\Pi|$, and the theorem is proved. □

Part III

Byzantine Atomic Broadcast to the Network Limit

8 Chop Chop

8.1 Overview

This Part focuses on Atomic Broadcast, the powerful consensus-equivalent primitive that has a set of processes agree on the (total) order of client-issued messages. Unlike Reliable Broadcast, which can only be applied to a relatively narrow class of useful problems (such as Asset Transfer [80]), Atomic Broadcast powers state machine replication (SMR) [62, 135], enabling decentralized universal computation in the face of arbitrary failures [104, 138].

Despite its versatility, however, Atomic Broadcast comes with fundamental bounds [59] and constraints [68], hindering its real-world performance despite decades of extensive research [10, 22, 36, 43, 49, 99, 108, 121, 155, 156, 159] and attention from industry, where SMR powers a myriad of blockchains and ledgers [8, 78, 98, 102, 107, 116, 143, 144, 145, 152, 153].

When deployed globally, seminal Atomic Broadcast implementations, such as BFT-SMaRt [22] and HotStuff [156], fail to stretch past a few thousand messages per second, three orders of magnitude short of the millions of requests per second collectively handled by the Internet's largest, centralized services (Figure 8.1). As we discussed in the introduction to this thesis, bringing Atomic Broadcast into the tens of millions of messages per second seems a necessary stepping stone towards planetary-scale, decentralized computation.

Towards line rate. While slow and expensive, ordering messages in Atomic Broadcast is amenable to *batching* [43]: order once, deliver in bulk. This observation motivated the development of *memory pool* (mempool) protocols [55, 70, 140], as initiated by Narwhal [55], designed to amortize ordering. This strategy proved effective, e.g., Bullshark [140] delivers in the order of 380,000 messages per second when accelerated by Narwhal. Despite this improvement, however, state-of-the-art batching still falls short of achieving *line rate*, i.e., matching the communication complexity of a protocol that does not ensure any ordering, authentication, or Byzantine resilience. In such a simplified setting, a server could simply deliver a sequence of application messages as it receives them from the network: b bits

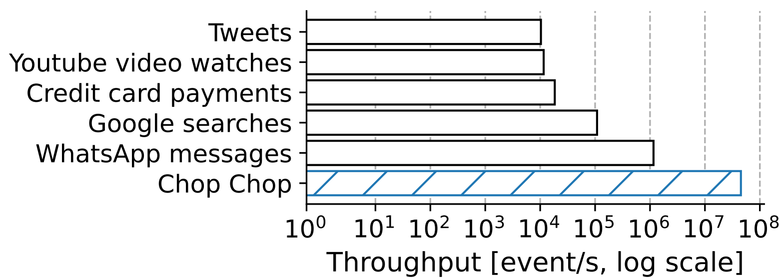


Figure 8.1: Throughput of Internet-scale services.

received, b bits delivered. Modern connections have enough bandwidth to receive tens of millions of application messages per second:[1] 2.5 orders of magnitude of gap still exist between Atomic Broadcast and unordered, unauthenticated dissemination. It is natural to ask if such a large gap is inherent to atomicity's unavoidable cost of ordering, authenticating and deduplicating messages. In this Part, we answer in the negative, accelerating Atomic Broadcast by a further two orders of magnitude with a system that performs within 8% of line-rate[2], even when handling 40 million requests per second.

Chop Chop. We present Chop Chop, a Byzantine Atomic Broadcast system. As a mempool, Chop Chop uses an underlying instance of Atomic Broadcast to order batches, aggregating messages to amortize costs. Classic methods of batching, however, fail to also amortize authentication and deduplication: each payload in a batch still has to carry an individual public key, signature and sequence number.

Chop Chop addresses this shortcoming with a new form of batches: *distilled batches*. Unlike a classic batch, a distilled batch contains condensed information that allows to authenticate and deduplicate its messages in bulk, much faster than in existing schemes. Distilled batches leverage the strong ordering of Atomic Broadcast to minimize redundant information.

Trustless brokers. Chop Chop produces distilled batches using a novel interactive protocol involving *brokers*, a layer of facilitating processes between clients and servers. Counterintuitively, distilled batches are faster for servers to receive and process, but expensive for brokers to produce: distillation is interactive and relies on expensive cryptographic operations.

Importantly, however, incorrectly distilled batches are visibly malformed. As such, brokers are *untrusted*: good brokers take load off the servers; bad ones cannot compromise the system's safety. Servers are exposed to every message in the system, bottleneck easily, and only a threshold of them can be compromised before the system loses safety. Brokers, instead, can

[1] Payloads as small as 12 bytes can have real-world applications (see Section 8.2.1). A 5 Gbit/s link can receive 52 millions such payloads per second.

[2] Up to some minor differences, line-rate can be considered a real-world measure of oracularity, the property we introduced in Part II to describe the ability of a distributed system to match the complexity of its centralized counterpart.

be spun up by anyone, outside of Chop Chop's security perimeter, to meet the load produced by clients.

Evaluation. We evaluate Chop Chop in a cross-cloud, geo-distributed environment including 320 medium-sized AWS EC2 machines and 64 OVH machines. We simulate up to 257 million clients and consider 12 experimental environments. Setting up each environment requires the installation of 13 TB of synthetic workload. A naive installation using `scp` from a single machine would take 68 hours. We designed `silk`, a one-to-many peer-to-peer file transfer tool optimized for high latency connections, to install the files in 30 minutes instead.

We compare Chop Chop's throughput and end-to-end latency against its baselines in multiple real-world scenarios including server failures, adverse network conditions, and applications running. In all scenarios, Chop Chop's throughput outperforms its closest competitor by up to two orders of magnitude, with no penalty in terms of latency. When put under stress, Chop Chop orders, authenticates and deduplicates upwards of 43,600,000 messages per second with a mean latency of 3.6 seconds. Except under the most adverse network conditions and proportions of faulty clients, Chop Chop still achieves millions of operations per second.

Applications. Unlike most Atomic Broadcast implementations [22, 55, 140, 156], Chop Chop does not offload authentication and deduplication to the application. This allows Chop Chop-based applications to focus entirely on their core logic without ever engaging in expensive, and easy to get wrong, cryptography. To showcase this, we implement three simple applications to evaluate on top of Chop Chop: a Payment system, an Auction house and an instance of the game "Pixel war". These three simple applications (300 lines of logic) work effectively with messages as small as 8 bytes, further underlying the communication overhead represented by public keys, signatures and sequence numbers in non-distilled systems. Both Payments and Pixel war inherit Chop Chop's throughput, respectively processing over 32 and 35 million operations per second. Even the Auction house, which is single-threaded, achieves 2.3 million operations per second. (These applications are meant as examples, and further optimization is beyond the scope of this Part.)

Contributions. We identify authentication and deduplication as the main bottlenecks of batched Atomic Broadcast; we introduce distilled batches to extend the amortizing properties of batching to authentication and deduplication; we present distillation, an interactive protocol to produce distilled batches, and identify the opportunity to offload it to an untrusted set of brokers; we implement Chop Chop, an Atomic Broadcast system taking advantage of distillation; we thoroughly evaluate Chop Chop, improving state-of-the-art Atomic Broadcast throughput by two orders of magnitude, maintaining near line-rate performance up to 40 million requests per second; we showcase Chop Chop through a Payment system, an Auction house and an instance of the "Pixel war" game, respectively achieving 32, 2.3 and 35 million operations per second.

Roadmap. Section 8.2 introduces Atomic Broadcast, discusses classic batching mechanisms and highlights the cost of authenticating and deduplicating messages in the resulting batches. Section 8.3 presents distilled batches and, for pedagogical purposes, introduces a simplified failure-free version of Chop Chop's protocol. Section 8.4 describes Chop Chop's fault-tolerant protocol in further detail. Section 8.5 discusses Chop Chop's implementation. Section 8.6 discusses Chop Chop's empirical evaluation, highlighting the challenges of such large scale experiments. We summarize related work in Section 8.7.

8.2 Atomic Broadcast

In an Atomic Broadcast system [45], *clients* broadcast messages that are delivered by *servers*.

Properties [34]. Correct servers deliver the same messages in the same order (*agreement*). Messages from correct clients are eventually delivered (*validity*). Spurious messages cannot be attributed to correct clients (*integrity*). No message is delivered more than once (*no duplication*).

8.2.1 Cost of Atomic Broadcast

Informally, Atomic Broadcast's most distinctive property, agreement, is also the most challenging to satisfy. Correct servers must coordinate to *order* messages without compromising liveness. A great deal of research effort has been put in developing ordering techniques, optimizing for latency [99, 114] or communication complexity [113, 124].

Integrity and no duplication, instead, allow for simple solutions. Clients can ensure integrity by *authenticating* their messages using digital signatures: servers simply ignore incorrectly authenticated messages. For no duplication, clients can tag each message with a strictly increasing *sequence number*: after ordering, servers discard old messages as replays.

Both techniques—we call them *classic authentication* and *classic sequencing*—are non-interactive, easy to implement, and agnostic of the protocol employed to order messages. Arguably due to the simplicity and effectiveness of classic authentication and sequencing, most Atomic Broadcast implementations overlook integrity and no duplication entirely: they offload authentication and sequencing to the application, focusing on the more challenging task of ordering.

Batching for ordering. Lacking an efficient technique to minimize its complexity, ordering could be Atomic Broadcast's main bottleneck.[3] The well-known strategy of *batching*, however, is both general and effective at amortizing the agreement cost of an Atomic Broadcast

[3] Byzantine Atomic Broadcast among n participants cannot be achieved with a bit complexity smaller than $\Theta(n^2)$ [59].

implementation [43, 138].

Broadly speaking, batching is orchestrated by a *broker* as follows [55].[4] Over a small window of time, the broker collects multiple client-issued messages in a batch, which it disseminates to the servers; the broker then submits to an underlying instance of Atomic Broadcast a cryptographic hash of the batch it collected; upon delivering the hash of a batch from Atomic Broadcast, a server retrieves the batch, and delivers to the application all the messages it contains. Because the size of a hash is constant, the cost of ordering a batch does not depend on its size: as batches become larger, the cost of ordering each message goes to zero. In practice, batching can effectively eliminate the cost of ordering in any real-world implementation of Atomic Broadcast.

Cost of integrity and no duplication. Batching does not efficiently uphold integrity and no duplication. Regardless of how many messages are batched together, the cost of classic authentication and sequencing stays constant: one public key, one signature and one sequence number for each message.

In practice, these costs dominate the computation and communication budget of a batched Atomic Broadcast system (see Section 8.3.2). On the one hand, signatures are among the most CPU-intensive items in the standard cryptographic toolbox, dwarfing in particular symmetric primitives such as hashes and ciphers. On the other, public keys, signatures and sequence numbers can easily account for the majority of a batch's size.

To illustrate these costs, consider the example of a payment system. A payment operation requires three fields: sender, recipient, and amount. Sender and recipient fit in 4 B each if the system serves less than 4 billion users. Amount needs 4 B for payments between 1 cent and 40 millions. Hence, a payment can be encoded in just 12 B. Using public keys to identify sender and recipient (2×32 B using Ed25519 [21, 90]) and attaching a signature (64 B) and a sequence number (8 B) to each message inflates payloads to 140 B. For payments, *91% of the bandwidth is spent on integrity and no duplication.*

8.2.2 Existing Mitigations

Chop Chop integrates the two following techniques to reduce the bandwidth and CPU cost of authentication.

Short identifiers. Repeated public keys consume a significant slice of a server's communication budget. A workaround is to have servers store public keys in an indexed *directory* [4]. Upon first joining the system, a client announces its public key via Atomic Broadcast to *sign up*. Upon delivering a sign-up message, a server appends the new public key to its directory.

[4] In the literature, servers usually play the role of brokers. As we discuss in Section 8.4, however, Chop Chop minimizes its load on the servers by offloading brokerage to a separate, trustless set of processes.

The same public key appears at the same position in the directory of all correct servers thanks to Atomic Broadcast's agreement. Having signed up, a client uses its position in the directory as identifier instead of its public key.

In the previous example of a payment system, using such identifiers reduces a payment size by 40%, from 140 B to 84 B. However, a signature per payment must still be transmitted.

Pooled signature verification. Authenticating a batch by verifying its signatures is a computationally intensive task for a server [43, 142]. However, Red Belly[53] and Mir [142] showed that not all servers need to authenticate all batches. Indeed, assuming at most f faulty servers, a broker optimistically asks only $f+1$ servers to authenticate a batch to be certain to reach at least one correct server. If $f+1$ servers do not reply by a timeout, the broker extends its request to f additional servers, thus reaching at least $f+1$ correct servers.

A correct server that authenticates a batch sends back to the broker a *witness shard*, i.e., a signed statement that the batch is correctly signed. The broker aggregates $f+1$ identical shards into a *witness*, which it sends to the other $2f$ servers. Because every witness contains at least one correct shard, the servers can trust the witness instead of verifying the batch.

Assuming $3f+1$ servers, this technique shaves up to two-thirds off the system's authentication complexity.

8.3 Distilled Batches

Chop Chop's main contribution is *distillation*, a set of techniques aimed at extending the amortizing properties of batches to authentication and sequencing.

Background: multi-signatures. Chop Chop makes use of multi-signature schemes [88] to authenticate batches. Secret keys produce signatures that can be verified against the corresponding public keys. Public keys and signatures, however, can be *aggregated*. Let $(p_1, r_1), \ldots, (p_n, r_n)$ be distinct key pairs, and $s_1, \ldots, s_n$ be signatures produced by $r_1, \ldots, r_n$ on the *same* message m: $p_1, \ldots, p_n$ (resp., $s_1, \ldots, s_n$) can be aggregated into a constant-sized aggregate public key p (resp., aggregate signature s).

Remarkably, s can be verified in constant time against p and m [25, 115]. Chop Chop uses BLS multi-signatures [25] which can be aggregated cheaply and non-interactively: even a non-signing process can compute p (resp., s) once provided with $p_1, \ldots, p_n$ (resp., $s_1, \ldots, s_n$) by computing a single multiplication over an elliptic curve.

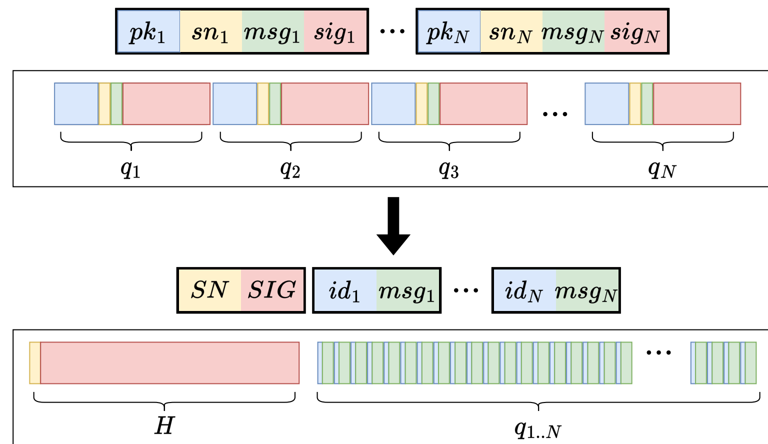


Figure 8.2: Full distillation in action. With classic authentication and sequencing, each payload q_i contains a public key pk_i, a sequence number sn_i, a message msg_i and a signature sig_i. In the fully distilled case, each q_i reduces to just id_i and msg_i: one header H, composed of one aggregate sequence number SN and one aggregate signature SIG, is sufficient for the entire batch. Bars are to scale if small messages are broadcast using Ed25519 for signatures and BLS12-381 for uncompressed multi-signatures: sn_i and SN are 8 B, msg_i is 8 B, pk_i is 32 B, sig_i is 64 B, SIG is 192 B.

8.3.1 Distillation at a Glance

In brief, distillation aims to produce *distilled batches*. A distilled batch has some of its signatures (resp., sequence numbers) replaced by an *aggregate signature* (resp., *aggregate sequence number*). When maximally successful, distillation produces a *fully distilled batch*, where all signatures (resp., sequence numbers) have been replaced by a *single* aggregate signature (resp., sequence number). As we discuss below, distilled batches are vastly cheaper for servers to receive and process. Figure 8.2 depicts the effect of distillation on a batch.

Full distillation (failure-free). For pedagogical purposes, we introduce distillation under the assumption that all processes are correct. We detail Chop Chop's fault-tolerant distillation techniques in Section 8.4.2, optimized and adapted to the Byzantine setting. As in the classic batching case, a set $\chi_1, \ldots, \chi_b$ of clients submit their messages $m_1, \ldots, m_b$ to a broker β. Each χ_i selects for its message m_i a sequence number k_i (greater than any sequence number it previously used), then sends (k_i, m_i) to β. Upon receiving all (k_i, m_i)-s, β computes the aggregate sequence number

$$k = \max_i k_i$$

then builds the *batch proposal*

$$B = [(x_1, k, m_1), \ldots, (x_b, k, m_b)]$$

where x_i is χ_i's numerical identifier in the system (see Section 8.2.2). β then sends B back to every χ_i. Upon receiving B, χ_i produces a multi-signature s_i for the hash $H(B)$ of B, which it sends back to β. Having collected all multi-signatures, β computes the aggregate signature

$$s = \prod_i s_i$$

In doing so, β obtains the fully distilled batch

$$\tilde{B} = [s, k, ((x_1, m_1), \ldots, (x_b, m_b))]$$

Upon receiving $\tilde{B}$, any server now can: compute B by inserting k between each (x_i, m_i); compute $H(B)$; use each x_i to retrieve χ_i's public key p_i from its directory; compute the aggregate public key

$$p = \prod_i p_i$$

and finally verify s against p and $H(B)$.

Distillation outcome. Having engaged with β to distill the batch, every χ_i multi-signs the *same* message $H(B)$ and updates its sequence number to the *same* k. This allows β to authenticate and sequence all of $\tilde{B}$ using s and k only.

Distillation safety. The proposed distillation protocol has no safety drawback. First, because (x_i, k, m_i) appears in B, χ_i still gets to authenticate m_i. Intuitively, χ_i's multi-signature on $H(B)$ publicly authenticates *whatever message in B is attributed to χ_i*, m_i in this case. Second, because $k \geq k_i$, k is still a valid sequence number for m_i. Sequence number distillation might cause χ_i to skip some sequence numbers whenever any χ_j issues some $k_j > k_i$. Contiguity of sequence numbers, however, is not a requirement for deduplication. As with classic sequencing, χ_i produces—and servers deliver—messages with strictly increasing sequence numbers; servers disregard all other messages as replays.

8.3.2 Distillation Microbenchmark

Having discussed how distilled batches are produced, we now estimate the significance of their effect by means of a back-of-the-envelope calculation and a simple microbenchmark on AWS. Consider a setting where 100 million clients broadcast 8-byte messages, e.g., to issue payments (see Section 8.2.1). We compare *classic authentication and sequencing*, where clients are identified by their public keys, messages are individually signed and sequenced, against *fully distilled batches* where clients are identified by a numerical identifier and each batch contains only one aggregated signature and sequence number. We use Ed25519 [90] for signatures (32 B public keys, 64 B signatures) and BLS12-381 [27] for multi-signatures (192 B uncompressed signatures). We use uncompressed BLS multi-signatures to save computation

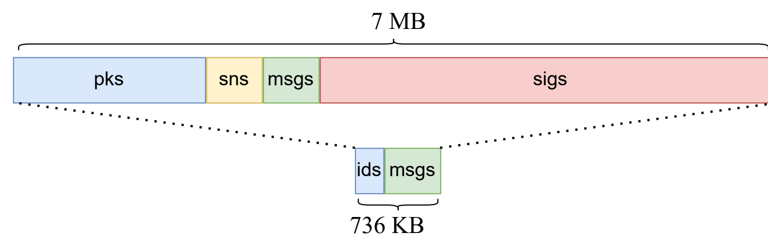


Figure 8.3: Full distillation of a batch of 65,536 payloads (sizes to scale). The aggregate signature and aggregate sequence number do not appear as a result of their small size.

time at the cost of storage space (96 B compressed vs. 192 B uncompressed).

Communication complexity. Payloads are 112 B per message in the classic case (32 B of public key, 8 B of sequence number, 8 B of message, 64 B of signature) vs. 11.5 B in the fully distilled case (28 bits = 3.5 B of identifier to represent 257M clients, 8 B of message). Assuming batches of 65,536 messages (Figure 8.3), classic batches are exactly 7 MB long, while fully distilled batches are 736 KB long including aggregate signature and sequence number.

Computation complexity. Running at maximum load, an Amazon EC2 c6i.8xlarge instance authenticates 16.2 ± 0.4 classic batches per second using Ed25519's batch verification for 65,536 signatures. The same machine authenticates 457.1 ± 0.3 fully distilled batches per second: each authentication requires the aggregation of 65,536 BLS12-381 public keys and the verification of one BLS12-381 multi-signature.

Summary. By the order-of-magnitude calculations above, fully distilled batches hold the promise to reduce the costs of authentication and sequencing by a factor 9.7 for network bandwidth, and 28.2 for CPU. Chop Chop aims to deliver on that promise for a real-world fault-tolerant system.

8.4 Chop Chop

This section overviews Chop Chop's architecture and protocol, and provides arguments for its correctness.

Overview. Chop Chop involves three types of processes (Figure 8.4): broadcasting clients, delivering servers and a layer of broadcast-facilitating brokers between them. Servers run an Atomic Broadcast instance among themselves, to which brokers submit messages. Chop Chop is agnostic to the implementation of Atomic Broadcast used by the servers. On top of the provided broker-to-server Atomic Broadcast, Chop Chop implements a much faster client-to-server Atomic Broadcast: clients submit messages to the servers, aided by brokers.

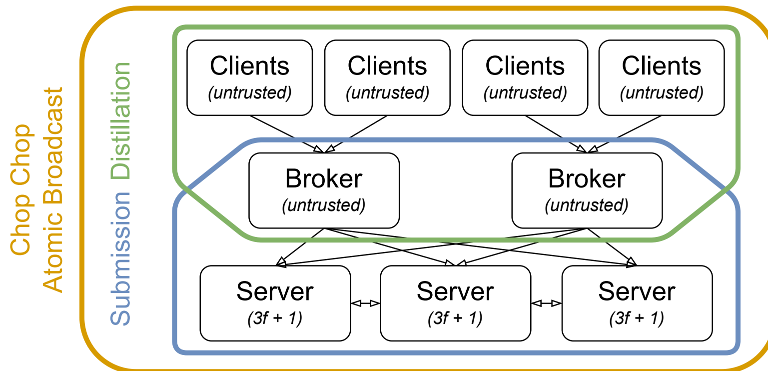


Figure 8.4: Chop Chop architecture.

Chop Chop's protocol unfolds in two phases: *distillation* (Section 8.4.2) and *submission* (Section 8.4.3). In the distillation phase, clients interact with a broker to gather their messages in a distilled batch (see Section 8.3). In the submission phase, the broker disseminates the distilled batch to the servers and submits the batch's hash to the server-run instance of Atomic Broadcast. Upon delivering its hash from Atomic Broadcast, servers retrieve the batch and deliver its messages. Chop Chop's contributions mainly focus on the distillation phase. Chop Chop's submission strategy closely resembles prior batch-based Atomic Broadcast implementations [55, 70, 140].

8.4.1 Architecture and Model

Chop Chop augments the architecture of a classic Atomic Broadcast, as described in Section 8.2, with novel brokers.

Clients and servers. *Clients* broadcast messages to a (distinct) set of *servers*. We assume that less than one third of servers can be faulty and behave in an arbitrary manner, i.e., be Byzantine [104], while all clients can be faulty. For simplicity, servers form a fixed set that is known by all correct processes at system startup. Chop Chop can be extended for reconfiguration thanks to its modular use of Atomic Broadcast [22, 103] (Figure 8.4). Clients issue messages after broadcasting their public keys to the system (see Section 8.2.2).

Brokers. We discussed in Section 8.3 how both classic and distilled batches are assembled by a broker. The role of brokers is traditionally taken by servers. Given the additional strain put on brokers by Chop Chop's interactive distillation protocol, however, having servers be brokers would result in a waste of scarce, trusted resources. Importantly, however, distillation is *trustless*. On the one hand, agreement rests entirely on Chop Chop's underlying Atomic Broadcast instance, for which brokers are only clients. On the other hand, as we argue in Sections 8.4.2 and 8.4.4, a faulty broker cannot compromise integrity or no duplication:

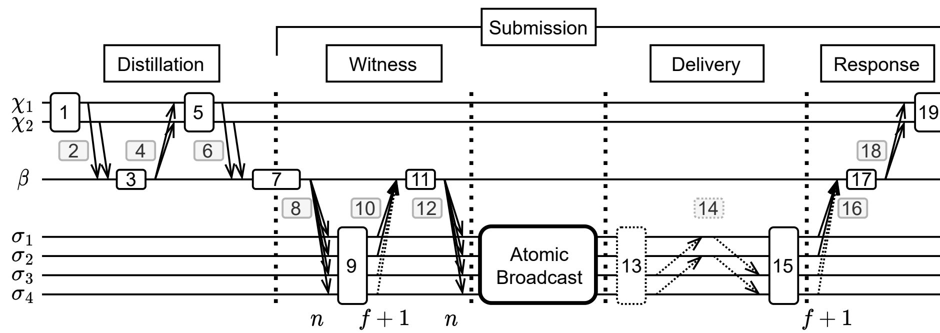


Figure 8.5: Overview of the Chop Chop protocol between two clients (χ_1, χ_2), a broker (β) and four servers (σ_1–σ_4). The protocol is comprised of 19 steps (#1–#19) and of an underlying instance of Atomic Broadcast such as BFT-SMaRt or HotStuff.

distilled batches are publicly authenticated, and correct clients cannot be tricked into using stale sequence numbers. Hence, *brokers need no trust*: a broker either does its job correctly or produces distilled batches that are visibly malformed, and easily discarded by all correct servers.

This observation is of paramount importance to the performance of Chop Chop: *because distillation is heavy but trustless, brokers should be distinct from servers.* Along with clients and servers, we thus assume a third, *independent set of brokers*, sitting between clients and servers, to accelerate Atomic Broadcast by assembling client messages in distilled batches. We assume that at least one broker is correct; the system loses liveness but not safety if all brokers are faulty.

Network. Chop Chop guarantees that the batches collected and submitted to servers by correct brokers are well-formed even in asynchrony, but achieves full distillation when the network is synchronous (see Section 8.4.2). Chop Chop inherits the network requirements of its underlying Atomic Broadcast.

8.4.2 Distillation Phase

We introduced in Section 8.3 a simplified, failure-free distillation protocol. This section describes how Chop Chop renders distillation tolerant to arbitrary failures and improves its performance via a sequence of improvements, each addressing a shortcoming of the simplified protocol. The complete fault-tolerant protocol of Chop Chop is depicted in Figure 8.5.

In the failure-free distillation protocol: clients $\chi_1, \ldots, \chi_b$ send their messages $m_1, \ldots, m_b$, with sequence numbers $k_1, \ldots, k_b$ (#2) to a broker β (Figure 8.5, #1); β identifies the maximum submitted sequence number k and builds a batch proposal $B = [(x_1, k, m_1), \ldots, (x_b, k, m_b)]$ (#3); β disseminates B to $\chi_1, \ldots, \chi_b$ (#4); each χ_i produces a multi-signature s_i on $H(B)$ (#5), which it sends to back β (#6); β aggregates $s_1, \ldots, s_n$ into an aggregate s, thus producing a fully

distilled batch $\tilde{B} = [s, k, ((x_1, m_1), \ldots, (x_b, m_b))]$ (#7).

Background: Merkle trees. Chop Chop uses Merkle trees [117] to hash batches. An l-element vector $z_1, \ldots, z_l$ is hashed into a *root* r, used as commitment. For each i, z_i's value can be proved by means of a proof of inclusion p_i, verifiable against r and z_i. Proofs of inclusions are $O(\log l)$ in size and are verified in $O(\log l)$ time.

What if a broker forges messages? A faulty β could try to falsely attribute to some χ_i a message $m'_i \neq m_i$. β could do so by replacing m_i with m'_i in B, then having χ_i sign $H(B)$, thus implicitly authenticating m'_i. This is easily fixed by having χ_i check that m_i correctly appears in B before signing $H(B)$.

Can a broker avoid sending the entire batch? A clear inefficiency of the simplified protocol is that β has to convey all of B back to each χ_i. This is fixed using Merkle trees. Upon assembling B, β computes the Merkle root r of B, along with the Merkle proof p_i for each (x_i, k, m_i) in B. Instead of sending B to all clients, β just sends r, k and p_i to each χ_i. Upon receiving r, k and p_i, χ_i checks p_i against r and (x_i, k, m_i), producing s_i on r only if the check succeeds. If χ_i signs r, then (x_i, k, m_i) is necessarily an element of B. Importantly, however, β could inject $(x_i, k, m'_i \neq m_i)$ somewhere else in B, while still providing χ_i only with the proof for (x_i, k, m_i). This is solved by having servers ignore every distilled batch where two or more messages are attributed to the same client. This way, if χ_i signs r, then either m_i is the only message in B attributed to χ_i, or $\tilde{B}$ is rejected by all servers as malformed: either way, integrity is upheld.

What if a client does not multi-sign? Under the assumption that $\chi_1, \ldots, \chi_b$ are correct, β can safely wait until it collects all $s_1, \ldots, s_b$. This policy is clearly flawed in the Byzantine setting: a single crashed client can prevent β from ever aggregating s. Furthermore, lacking an assumption of synchrony, β cannot exclude from $\tilde{B}$ those clients that do not sign r by some timeout: consistently slow clients would always be excluded, and validity would be lost. This issue is fixed by the fallback mechanism introduced in the following.

Fault-tolerant distillation. Upon first sending (k_i, m_i) to β (#2), χ_i also sends an individual, non-aggregable signature t_i for (x_i, k_i, m_i), which β stores. β then waits for s_i-s on r until either all s_i-s are collected, or a timeout expires. For every s_i that ends up missing, due to χ_i being crashed or delayed, β attaches (k_i, t_i) to $\tilde{B}$. Upon receiving $\tilde{B}$, a server first checks each individual signature t_i against the corresponding (x_i, k_i, m_i). The server then checks s against the public keys of the clients for which an individual signature t_i was not given, i.e., the public keys of all clients that signed r in time.

In summary: fast, correct clients who successfully produce their s_i-s in time authenticate their message by multi-signing r; slow or crashed clients still get their messages through,

individually authenticated by the t_i-s that they originally produced. Full distillation is achieved whenever the network is synchronous and all clients are correct, which we argue is the case in practice for the majority of a system's lifetime. When the network is asynchronous, however, a fraction of clients might fail to produce their s_i in time, resulting in a *partially distilled batch*. At the limit where all clients fail to sign r in time, $\tilde{B}$ reduces to a classic batch, degrading server-side performance to pre-distillation levels. We underline that safety and liveness are preserved regardless of synchrony.

What if a broker replays messages? A problem introduced by the last fix is that χ_i authenticates both k_i and k as sequence numbers for m_i, allowing a faulty β to play m_i twice, hence breaking Atomic Broadcast's no duplication. This is fixed by having each client engage in the broadcast of only one message at a time. This way, while β can replay m_i, it can only do so consecutively: all sequence numbers χ_i authenticates for m_i belong to a range that does not contain sequence numbers for any other message $m_{i' \neq i}$ issued by χ_i.

This observation is key to the following fix: along with the last sequence number $\bar{k}_\chi$ each client χ used, a correct server σ stores the last message $\bar{m}_\chi$ that χ broadcast; upon ordering a message m with sequence number k from χ, σ delivers m if and only if $k > \bar{k}_\chi$ and $m \neq \bar{m}_\chi$. In doing so, σ discards all consecutive replays of $\bar{m}_\chi$, thus preventing replays in general.

What if a client broadcasts too frequently? The last fix relies on clients broadcasting one message at a time. Depending on latency, a client broadcasting too frequently might accrue an ever-growing queue of pending messages. This issue is fixed by flushing application messages in bursts, akin to Nagle's buffering algorithm for TCP.

What if a client submits the largest possible sequence number? Assuming that a finite number of bits (e.g., 64) are allocated to representing sequence numbers, a faulty client χ_m could set its k_m to the largest possible sequence number k_{max} (e.g., $2^{64} - 1$). In doing so, χ_m would force all other χ_i-s to update their sequence number to k_{max}. Since correct clients only use strictly increasing sequence numbers, no χ_i could ever broadcast again: sequence numbers would run out. Proving the *legitimacy* of sequence numbers fixes this issue.

Legitimate sequence numbers. By the rule that we established, no more than one message from the same client can appear in the same batch. Moreover, correct clients always tag their messages with the smallest sequence number they have not yet used, i.e., the largest they have used plus one. By induction, we then have that unless some client misbehaves, no client ever needs to use a sequence number larger than the number of batches ever delivered by the servers: the largest sequence number any client submits to the very first batch is 0, therefore no client submits a sequence number larger than 1 to the second batch, and so on. This observation allows us to define as *legitimate* any sequence number smaller than the number

of batches servers have delivered at any given time.

Legitimacy proofs. This definition of legitimacy allows for the generation of *legitimacy proofs*: upon delivering the n-th batch, a server publicly states so with a signature. By collecting $f+1$ server signatures stating that the n-th batch was delivered into a certificate l_n, any process can publicly prove that any sequence number smaller than n is legitimate.

Upon initially submitting k_i (#2), χ_i also sends to β a certificate l_n, for any $n > k_i$; β ignores client submissions that lack such certificate, except when $k_i = 0$ since no certificate is needed. Upon sending k back to all χ_i-s (#4), β attaches the highest $l_{\hat{n}}$ it collected: $l_{\hat{n}}$ proves that k is legitimate since $\hat{n} > k$. χ_i signs r (#5) only if k is proved legitimate by $l_{\hat{n}}$.

This technique ensures correct clients always use legitimate sequence numbers. Because legitimate sequence numbers grow only with the number of batches delivered by the servers, no correct client is forced to skip too far ahead, compromising its own liveness.

What if a broker crashes? If β fails to engage in the protocol, each χ_i can submit its message to any other broker.

8.4.3 Submission Phase

The submission phase ensures that all servers efficiently deliver a distilled batch, and that all broadcasting clients receive a proof that their messages were delivered.

Witness. Having gathered a distilled batch $\tilde{B}$ (#7), β moves on to have $f+1$ servers signs a *witness shard* for $\tilde{B}$. In signing a witness shard for $\tilde{B}$, a server σ simultaneously makes two statements. First, $\tilde{B}$ is *well-formed*: σ successfully verified $\tilde{B}$'s signatures and found all messages in $\tilde{B}$ to have a different sender. Second, $\tilde{B}$ is *retrievable*: σ stores $\tilde{B}$ and makes it available for retrieval, should any other server need it. We call a *witness* for $\tilde{B}$ the aggregation of $f+1$ witness shards for $\tilde{B}$. Because any set of $f+1$ processes includes a correct process, when presented with a witness for $\tilde{B}$ any server can trust $\tilde{B}$ to be well-formed and retrievable.

As discussed in Section 8.2.2, witnesses optimize server-side computation. Only $f+1$ servers need to engage in the expensive checks required to safely witness $\tilde{B}$. All other servers can trust $\tilde{B}$'s witness, saving trusted CPU resources.

In order to collect a witness for $\tilde{B}$, β sends $\tilde{B}$ to all servers (#8). Optimistically, β asks only $f+1$ servers to sign a witness shard for $\tilde{B}$, progressively extending its request to $2f+1$ servers upon expiration of suitable timeouts. Upon receiving $\tilde{B}$ (#9), a correct server σ stores $\tilde{B}$. If asked to witness $\tilde{B}$, σ checks that $\tilde{B}$ is well-formed and sends back to β its witness shard for $\tilde{B}$ (#10). β collects and aggregates $f+1$ shards into a witness for $\tilde{B}$ (#11), then submits $\tilde{B}$'s hash and witness to the server-run Atomic Broadcast (#12).

Delivery. Upon delivering $\tilde{B}$'s hash and witness from Atomic Broadcast (#13), a correct server σ retrieves $\tilde{B}$, either from its local storage (if it directly received $\tilde{B}$ from β at #8) or from another server (#14). Because $\tilde{B}$ is retrievable, σ is guaranteed to eventually find a server to pull $\tilde{B}$ from. Having retrieved $\tilde{B}$ (#15), σ delivers all non-duplicate messages in $\tilde{B}$ (see Section 8.4.2 for how σ detects duplicates).

Response. Finally, σ signs a *delivery certificate*, listing the messages in $\tilde{B}$ that σ delivered. σ sends its signature back to β (#16). By agreement of Atomic Broadcast, all correct servers deliver the same subset of messages in $\tilde{B}$. As such, β is guaranteed to eventually collect $f+1$ signatures on the same delivery certificate (#17). Upon doing so, β distributes a copy of $\tilde{B}$'s delivery certificate to $\chi_1, \ldots, \chi_b$ (#18). Armed with $\tilde{B}$'s delivery certificate, a correct χ_i can publicly prove the delivery of m_i (#19) and safely broadcast its next message.

8.4.4 Correctness

This section summarizes Chop Chop's correctness analysis.

Safety

The safety of Chop Chop is given by its agreement, integrity and no duplication properties (see Section 8.2).

Agreement. Chop Chop inherits agreement from its underlying, server-run instance of Atomic Broadcast. A correct server delivers messages only upon delivering the hash of a batch from the server-run Atomic Broadcast. Upon doing so, a correct server retrieves the full batch, checks its hash, and delivers all its messages in order of appearance. All correct servers deliver the same messages in the same order assuming cryptographic hashes are collision-resistant.

Integrity. A correct server only delivers messages included in a batch witnessed by $f+1$ servers, i.e., by at least one correct server. A correct server witnesses a batch only if: no more than one message in the batch is attributed to the same client; every client in the batch authenticates its message with a signature or the root of the batch's Merkle tree with a multi-signature. A correct client multi-signs the root of a batch's Merkle tree only upon receiving a proof of the inclusion of its message in the batch. As such, if a correct client multi-signs the root of a batch's Merkle tree, either the batch contains only the client's intended message or it is not witnessed. In summary, a correct server delivers a message m from a correct client χ only if χ broadcast m.

No duplication. A correct client only broadcasts one message at a time. As such, while the client might attach multiple sequence numbers to the same message (different brokers may propose different aggregate sequence numbers for the client to authenticate) the sequence numbers the client attaches to each message belong to distinct ranges. A correct server delivers client messages only in increasing order of sequence number, and ignores repeated messages. This means that a correct server delivers at most one message from each sequence number range. In summary, no server delivers a correct client's message more than once.

Liveness

The liveness of Chop Chop is given by its validity property.

Validity. If a correct client submits its message to a correct broker, the message is guaranteed to eventually be delivered by all correct servers: even if the client fails to engage in distillation in a timely manner, its message is still included in a batch which gets disseminated, witnessed and delivered by all correct servers. Faulty brokers can clearly refuse to service (specific) clients. Upon expiration of a suitable timeout, however, a correct client submits its message to a different broker. As we assume that at least one broker is correct, all correct clients are eventually guaranteed to find a correct broker and get their messages delivered by all correct servers.

Other Attacks

As we outlined in Section 8.4.4, Chop Chop satisfies all properties of Atomic Broadcast. In this section, we consider other attacks an adversary might deal to impair Atomic Broadcast's performance and fairness [93] in Chop Chop.

Denial of service. A faulty broker may refuse to service clients, thus forcing them to fall back on other brokers, increasing latency. A faulty broker may also submit deliberately non-distilled batches to servers to force them to waste trusted resources to receive and verify individual signatures. While handling DoS is beyond the scope of this Part, Chop Chop is amenable to accountability mechanisms [83]. Brokers could be asked to stake resource to join the system. Correct, high-performance brokers could be rewarded, akin to gas fees in Ethereum [153]. Brokers that accrue a reputation of misbehavior or slowness could be banned and lose their initial stake.

Front-running. A faulty broker might impact fairness by front-running messages of interest [54, 160]. While front-running resistance is beyond the scope of this Part, Chop Chop is compatible as-is with existing mechanisms to mitigate or prevent front-running, most notably schemes that have clients submit encrypted messages whose content is revealed only after

delivery [119, 158]. Importantly, these encrypt-order-reveal schemes could be selectively employed only for those messages that are vulnerable to front-runs, e.g., messages used for stock trading [128]. Maintaining Chop Chop's throughput while providing quorum-enforced fairness for every message [159] opens a valuable future avenue of research.

8.5 Implementation Details

A straightforward implementation of the protocol we presented in Section 8.4 would not achieve the throughput and latency we observe in Section 8.6. In this section, we discuss some of the techniques and optimizations required on the way to practically achieving Chop Chop's full potential. (Many optimizations are however left out due to space constraints).

Code. Chop Chop is implemented in Rust, totaling 8,900 lines of code. The main libraries Chop Chop depends on are: `tokio` 1.12 for an asynchronous, event-based runtime; `rayon` 1.5 for worker-based parallel computation; `serde` 1.0 for serialization and deserialization; `blake3` 1.0 for cryptographic hashes; `ed25519-dalek` 1.2 for EdDSA signatures on Curve25519 [90]; `blst` 0.3.5 for multi-signatures on the BLS12-381 curve [27]. Chop Chop also depends on in-house libraries: `talk` (9,800 lines of code) for basic distributed computing and high-level networking and cryptography; `zebra` (7,100 lines of code) for Merkle-tree based data structures.

8.5.1 Broker

The goal of a Chop Chop broker is to produce batches as distilled as possible (to minimize server load), as large as possible (to amortize ordering), and as quickly as possible (to minimize latency). Our target is for a broker to assemble one fully distilled batch of 65,536 messages (736 KB, see Figure 8.3) per second, with a 1 second distillation timeout.

Reliable UDP. Short-lived TCP connections between broker and clients are easier to work with, but unfeasible for the broker to handle. Assuming an end-to-end broadcast time of up to 10 seconds, the broker would need to maintain upwards of 600,000 simultaneous TCP connections, which preliminary tests immediately proved unfeasible on the hardware we have access to. This makes UDP the only option for client-broker communication. However, UDP lacks the reliability properties of TCP, and tests showed non-negligible packet loss even within the same AWS EC2 availability zone. As we discussed in Section 8.4.2, message loss immediately translates to partial distillation. We address this issue by means of an in-house, ACK-based, message retransmission protocol based on UDP that also smoothens the rate of outgoing packets.

EdDSA batch verification. To avoid spoofing, all client messages are authenticated with signatures. At the target rate, however, individually verifying each signature is unfeasible for a broker. Luckily, `ed25519-dalek` allows for more efficient batched verification. A broker buffers the client messages it receives and authenticates them in batches.

Tree-search invalid multi-signatures. Clients contributing to the same batch produce matching multi-signatures for the batch's root. At the target rate the broker cannot independently verify each multi-signature. We tackle this problem by gathering multiple matching multi-signatures on the leaves of a binary tree: internal nodes aggregate their children. For each tree, the broker verifies the root multi-signature, recurring only on the children of an invalid parent. This allows to identify invalid multi-signatures in logarithmic time while enabling batched verification in the good case.

Caching legitimacy proofs. Clients justify their sequence numbers with legitimacy proofs. Again, the broker cannot verify each proof in time. We address this problem by having the broker verify a legitimacy proof only if higher than the highest it previously observed. As a result, a faulty client might get away with submitting an invalid legitimacy proof but, importantly, not an illegitimate sequence number.

8.5.2 Server

The goal of a Chop Chop server is to process distilled batches as quickly as possible without overflowing its memory.

Batch garbage collection. Servers update each other on which batches they delivered. A server garbage-collects a batch, both messages and metadata, as soon as it is delivered by all other servers. We underline that, even if a single server fails to deliver a batch, the others cannot garbage-collect it as the slow server might be correct. This is an inherent limitation of Atomic Broadcast: agreement without synchrony can be ensured only in the infinite-memory model.

Identifier-sorted batching. No two messages from the same client must appear in the same batch. To simplify processing, brokers sort the messages in a batch by client identifier. Servers reject batches whose identifiers are not strictly increasing, thus verifying that all identifiers are distinct in constant size and in linear time. Sorting messages by identifier also enables parallel deduplication: messages are split by identifier range, chunks are deduplicated independently.

8.6 Evaluation

We evaluate Chop Chop focusing on the following research questions (RQs): What workload can Chop Chop sustain (Section 8.6.3)? What are the benefits of Chop Chop's distillation (Section 8.6.4)? How does Chop Chop scale to different numbers of servers (Section 8.6.5)? How efficiently does Chop Chop use resources overall (Section 8.6.6)? How does Chop Chop perform under adverse conditions, such as server failures (Section 8.6.7)? What performance can applications achieve using Chop Chop (Section 8.6.8)?

8.6.1 Baselines

We compare Chop Chop against four baselines:

- HotStuff [156]: an Atomic Broadcast protocol designed for high-throughput (written in C++);
- BFT-SMaRt [22]: an Atomic Broadcast protocol, similar to PBFT [43], designed for low-latency (written in Java);
- Narwhal-Bullshark: the DAG-based Atomic Broadcast protocol Bullshark [140] with the state-of-the-art high-throughput mempool Narwhal [55] (written in Rust);
- Narwhal-Bullshark-sig: akin to Narwhal-Bullshark but with Narwhal modified to authenticate messages, thus matching Chop Chop's guarantees.

We deploy Chop Chop with two distinct underlying Atomic Broadcast protocols (Figure 8.5): HotStuff and BFT-SMaRt.

HotStuff and BFT-SMaRt. Evaluating HotStuff and BFT-SMaRt allows us to assess the base performance of an Atomic Broadcast protocol and determine how much acceleration Chop Chop provides. We evaluate Chop Chop on top of the same implementations of HotStuff[5] and BFT-SMaRt[6] we benchmark against. These implementations are production-ready and do not use state-of-the-art mempool protocols, only some basic form of batching. When evaluated stand-alone, each message in these systems includes 80 B of header composed of a client identifier (8 B), a sequence number (8 B), and a signature (64 B) verified by the servers. Both systems use batches of 400 messages, i.e., of 34.4 KB.

Narwhal-Bullshark. As a state-of-the-art mempool, Narwhal[7] is a close point of comparison for Chop Chop. Servers in Narwhal scale out following a primary-workers model: each server is paired with one or several workers into a server group. Similarly to Chop Chop, Narwhal greatly accelerates its underlying Atomic Broadcast (here, Bullshark). Unlike Chop Chop,

[5]Authors' implementation of HotStuff in C++: https://github.com/hot-stuff/libhotstuff
[6]Authors' implementation of BFT-SMaRt in Java: https://github.com/bft-smart/library
[7]Authors' implementation of Bullshark and Narwhal in Rust: https://github.com/asonnino/narwhal/tree/bullshark

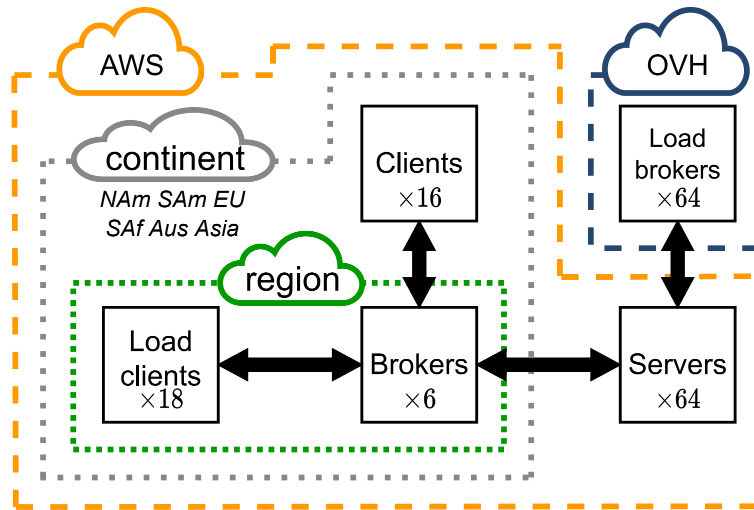


Figure 8.6: Cross-cloud deployment summary.

however, Narwhal leaves the responsibility of authenticating and deduplicating messages to the application.

Narwhal-Bullshark-sig. For a better comparison, we also benchmark Narwhal-Bullshark-sig: Narwhal-Bullshark where messages are authenticated by Narwhal in a state-of-the-art way, i.e., using batched, multi-core Ed25519 signature verification. Each message includes an 80 B header as for HotStuff and BFT-SMaRt. As for Narwhal-Bullshark, the remaining parameters are the default ones, e.g., 500 KB batches.

8.6.2 Setup

Unless otherwise specified—in Sections 8.6.5 and 8.6.6—the Chop Chop benchmarks involve 64 c6i.8xlarge AWS servers, of 32 Intel vCPUs each, geo-distributed across 14 regions. Brokers assemble, and servers process, batches of 65,536 messages. Each message is 8 B in length, resulting in 736 KB batches (Figure 8.3). Baselines always use the same set of server machines as their Chop Chop counterpart. All experiments run with maximum resilience, e.g., the system survives 21 faulty servers out of 64. Figure 8.6 overviews the used deployment.

Matching trusted vs. total resources. Unlike its baselines, Chop Chop leverages *untrusted* resources, brokers, to boost its performance. Lacking a well-defined conversion between trusted and untrusted resources, two extremes can be taken to compare Chop Chop with its baselines: we can either match trusted resources, e.g., same number of Chop Chop servers as Narwhal workers, or match total resources, e.g., same number of servers and brokers in Chop Chop as workers in Narwhal.

Intuitively, the first approach considers untrusted resources to be free while the second considers untrusted resources to be as costly as trusted resources. We use the first approach in Sections 8.6.3 to 8.6.5, 8.6.7 and 8.6.8 to stress Chop Chop, provisioning the system with enough brokers to bottleneck servers. We use the second approach in Section 8.6.6 to assess how efficiently Chop Chop uses its hardware resources, trusted or not.

Load clients and load brokers. We show in Section 8.6.3 that Chop Chop servers handle up to 43.6 million operations per second with an average latency of 3.6 seconds. To produce this level of workload, a real-world deployment would require over 700 brokers, each handling around 200,000 clients broadcasting back-to-back thus totaling hundreds of millions of machines. As we cannot experiment at such a scale, we introduce two new actors: *load clients* and *load brokers*.[8]

Load clients connect to brokers and simulate thousands of concurrent client requests. Most system evaluation typically use this approach to stress the system and measure latency. However, we explicitly separate clients from load clients in this evaluation. Clients run on very small machines—less powerful than most smartphones—to provide more accurate end-to-end latency measurements. We similarly split clients from load clients in all baseline runs.

Load brokers are unique to Chop Chop. Even using load clients, we could not deploy enough brokers to bottleneck Chop Chop's servers. Load brokers work around this limitation, submitting batches of pre-generated messages directly to the servers. Free from interactions with clients and expensive cryptography, a load broker puts on the servers a load equivalent to that of tens of brokers working at full capacity.

Using load clients and load brokers, we manage to show that brokers can quickly generate large batches of messages, and servers can process large numbers of batches.

Cross-cloud deployment. All servers are deployed on AWS, balanced across 14 regions: Cape Town, São Paulo, Bahrain, Canada, Frankfurt, Northern Virginia, Northern California, Stockholm, Ohio, Milan, Oregon, Ireland, London, and Paris. For system sizes of 8 in Section 8.6.5, we distribute servers across the first 8 regions from the list, which constitute the most adversarial setup with the highest pairwise latency.

Load brokers are placed in a separate cloud provider, OVH, for two purposes. First, it provides a better representation of Internet load than a single-cloud deployment. AWS operates under its own AS so any AS peering bottlenecks would be bypassed by an AWS-only deployment. Second, OVH is one of the few cloud providers with enough peering with AWS to stress Chop Chop without charging for egress bandwidth, saving us from using AWS' costly bandwidth. The final cost amounted to 25,000 USD in AWS credits. Using OVH saved us more than 70,000

[8] Throughout the remainder of this section, "brokers" and "clients" denote real brokers and real clients. The term "load" is always used explicitly.

USD since each of Chop Chop's data point on a figure would have cost 1,700 USD in AWS egress bandwidth—21 TB at 0.08 USD per GB ≈ 1,700 USD.

For all experiments, we deploy one broker in each continent (Cape Town, São Paulo, Tokyo, Sydney, Frankfurt, and Northern Virginia) and one client in each of the 14 regions above, plus Tokyo and Sydney. Clients connect to their nearest broker. We configure the network for geo-distribution and high load, e.g., TCP buffer sizes [9] and UDP parameters.

All baselines run on the same parameters. For Narwhal-Bullshark, we collocate each server with one of the workers in its server group. We reproduced Narwhal-Bullshark's original experiments [140] and matched the results.

Hardware. All servers, brokers and load clients run on c6i.8xlarge machines with an Intel Xeon Platinum 8375C (32 virtual CPUs, 16 physical cores, 2.9 GHz baseline, 3.5 GHz turbo), 64 GB of memory and 12.5 Gb/s of bandwidth. We selected these machines since they provide good performance and are in the same "commodity" price range as those chosen initially for Chop Chop's main baseline: Narwhal-Bullshark. Clients run on t3.small machines: 2 vCPUs, 1 physical core, 2 GB of memory, and up to 5 Gb/s bandwidth—of which they use less than 1 KB/s. All machines run Linux Ubuntu 20.04 LTS on the AWS patched version of the Linux kernel 5.15.0, except for the load brokers on OVH which run on Linux kernel 5.4.0—the same kernel was not available.

Challenges. The most significant evaluation challenges arose from the scale of the targeted deployment. The setup and orchestration alone required simultaneous handling of up to 320 machines across two different cloud providers and 25 regions, as well as transferring 13TB of files—mostly public keys and pre-generated batches—for each of the 12 setups. To handle this, we developed a new command-line tool to efficiently deploy distributed systems: `silk`. Among other things, we use `silk` for peer-to-peer-style file transfer over aggregated TCP connections, and for grouped process control. With `silk`, transferring all files from a single machine takes around 30 minutes, compared to 68 hours with `scp`.

Additional challenges came from the real-world nature of the targeted deployment. First, the connection between OVH and AWS's Asia and Pacific regions was particularly unstable at certain times of day especially when close to saturation. For example, Tokyo's connection was frequently degraded between 3pm and 5pm UTC. Second, the performance of some machines sometimes deviated from their specifications. As an example, in a setup size of 64, we observed around 2 machines operating with a 10% lower CPU turbo clock rate than specified. Considering these variations, we increased the number of servers a broker initially asks for witness shards (see Section 8.4.3) by a margin, e.g., $f+5$ instead of $f+1$. This improves system stability—i.e., lower latency variability—while slightly reducing maximum throughput.

[9] IBM's TCP Tuning Guide: https://www.ibm.com/docs/en/linux-on-systems?topic=recommendations-network-performance-tuning

Unless otherwise specified, we set the margin to 4 in all experiments, i.e., $f+5$.

Plots. Every data point is the mean of 5 runs of 2 minutes each (after excluding warmup and cooldown, the relevant cross-section is at least 1 minute). All plots further depict one standard deviation from the mean using either colored shaded areas or black error bars (which may be too small to notice).

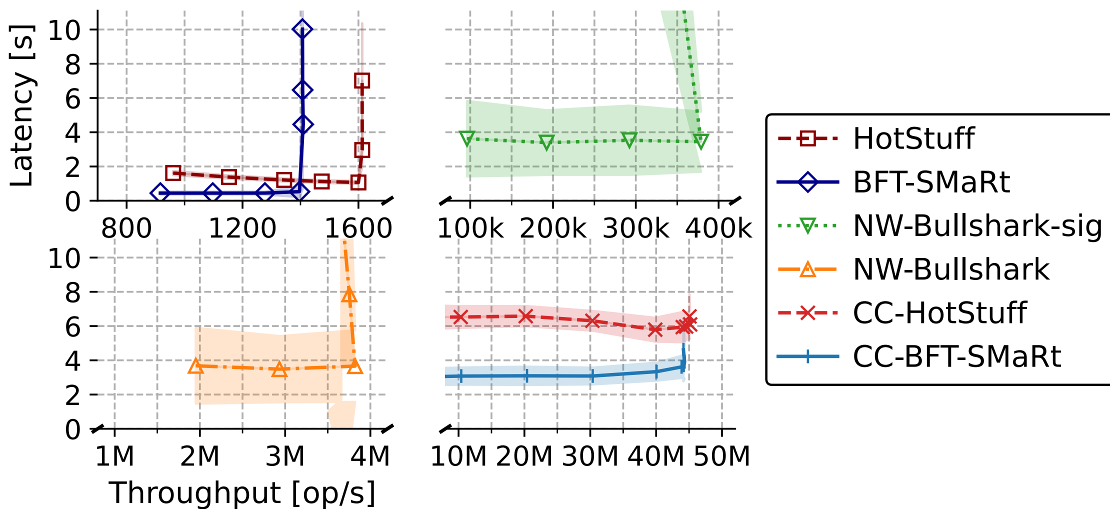


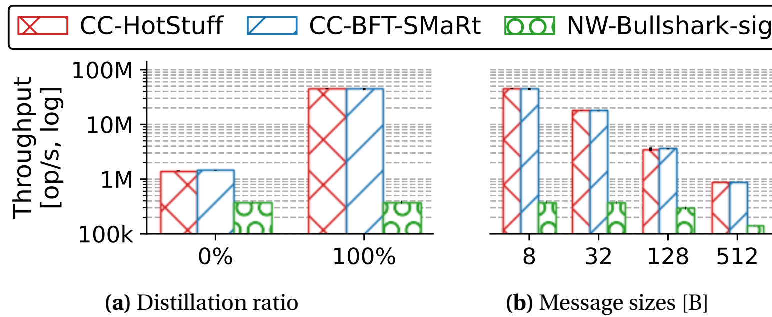
Figure 8.7: Throughput-latency of Chop Chop and of notable Atomic Broadcast systems under various input rates.

8.6.3 RQ1 – Load Handling

Figure 8.7 shows the latency and throughput of Chop Chop and all its baselines for various input rates of 8 B messages. The variability is represented using shaded areas.

Baselines. Both BFT-SMaRt and HotStuff showcase stable performances under low loads, respectively achieving around 1,400 and 1,600 operations per second. BFT-SMaRt's latency is consistently better than HotStuff's up to its inflection point (0.45–0.53 s vs. 1.2–1.6 s). We measure up to 3.8M op/s for Narwhal-Bullshark and up to 382k op/s for Narwhal-Bullshark-sig. The difference in respective throughput highlights the cost of authentication for servers: verifying signatures reduces the throughput of Narwhal-Bullshark by one order of magnitude. We observe a latency of around 3.6 s for both Narwhal-Bullshark and Narwhal-Bullshark-sig.

Chop Chop. Chop Chop achieves close to 44M op/s while running on top of both HotStuff and BFT-SMaRt. Chop Chop's latency range is 3.0–3.6 s with BFT-SMaRt and 5.8–6.5 s with HotStuff. Notably, the latency of Chop Chop-HotStuff decreases under high load. This is due to the internal batching mechanism of the HotStuff implementation: buffers fill faster under



(a) Distillation ratio

(b) Message sizes [B]

Figure 8.8: Throughput of Chop Chop and authenticated Narwhal with Bullshark (log scale) when (a) Chop Chop has no distillation and with (b) varying message size.

higher load, thus avoiding timeouts. This has an immediate impact on Chop Chop, which feeds HotStuff at a low rate: HotStuff alone accounts for over 60% of Chop Chop-HotStuff's overall latency. BFT-SMaRt makes a better fit for Chop Chop, as its throughput is sufficient for Chop Chop's needs, and its latency is lower than HotStuff's.

Mempools' trade-off. In comparison to BFT-SMaRt and HotStuff, Chop Chop trades latency in favor of throughput. This trade-off is mostly explained by batching and distillation. When assembling a batch, a broker has to wait twice: once to collect enough messages to fill a batch, and once to collect all multi-signatures from clients engaging in distillation. We set both waits' timeout to 1 second. Notably, Narwhal-Bullshark seems to incur a similar latency cost, as Chop Chop's latency approximately matches that of Narwhal-Bullshark, even though Chop Chop needs an extra round trip between clients and broker (Figure 8.5, #4–#6).

8.6.4 RQ2 – Distillation Benefits

We showcase the benefits of distillation by: evaluating throughput with and without distillation, evaluating distillation for messages of different sizes, and observing the impact of distillation on network bandwidth to achieve line rate.

Distillation vs. mitigations. Along with distillation, Chop Chop makes use of two techniques available in the literature to mitigate the cost of Atomic Broadcast's authentication: short identifiers and pooled signature verification (see Section 8.2.2).

Figure 8.8a breaks down Chop Chop's throughput, measuring how significantly distillation alone contributes to Chop Chop's performance. When no message is distilled, Chop Chop's servers bottleneck at 1.5M op/s, 3.9× higher than Narwhal-Bullshark-sig. This result is in line with both systems bottlenecking on server CPU, as the technique employed by Chop Chop to mitigate authentication complexity has only one third of the servers verify each client signature. (We conjecture that the additional 1.3× factor may be owed to engineering

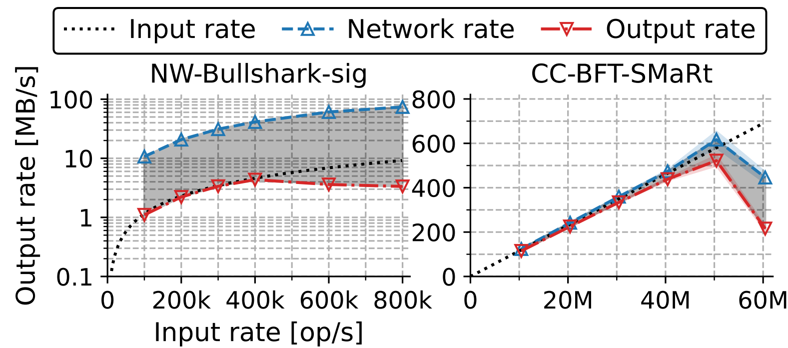


Figure 8.9: Throughput efficiency of authenticated Narwhal with Bullshark (left, log scale) and Chop Chop with BFT-SMaRt (right, linear scale) with various input rates.

differences.) When batches are fully distilled, Chop Chop's throughput grows to 44M op/s, accounting for the additional 29-fold boost to Chop Chop's performance.

Distillation for larger messages. Figure 8.8b illustrates Chop Chop's maximum throughput for message sizes of 8 B to 512 B which may be relevant to applications that cannot work around smaller message sizes, e.g., many smart contracts. Chop Chop's throughput is similar with BFT-SMaRt and HotStuff, decreasing at an approximately 1-to-1 ratio as the message size increases: 44.3M op/s for 8 B, 17.6M op/s for 32 B, 3.5M op/s for 128 B and 890k op/s for 512 B.

This is in line with expectations. As we discuss in Section 8.3.2, a server should receive $\sim b$ bytes in order to deliver a b-bytes message in a large, fully distilled batch, as full distillation amortizes to zero the communication cost of authenticating and sequencing each message. For 8 B messages, servers encounter a CPU bottleneck slightly before the link between load brokers and servers is saturated. This explains why the throughput decreases only 2.52× when messages grow to 32 B: all remaining server-bound bandwidth is used to convey messages (as messages are larger) while the load on server CPUs is reduced (as less messages are delivered overall). The system remains communication-bottlenecked as the size of the messages increases, and throughput starts decreasing linearly with message size, e.g., Chop Chop's throughput for 512 B messages is 4.00× smaller than for 128 B.

By contrast, Narwhal-Bullshark-sig bottlenecks on server CPUs longer, due to signature verification, maintaining a stable throughput until 512 B messages finally fill server links. Overall, Narwhal-Bullshark-sig's throughput only decreases from 382k op/s for 8 B messages to 142k op/s for 512 B messages, which matches their non-authenticated evaluation with 512 B messages. The gap between Chop Chop and Narwhal-Bullshark-sig at 512 B messages can be mostly attributed to Chop Chop's more efficient use of server bandwidth: unlike Narwhal, Chop Chop offloads the dissemination of batches to external brokers. Narwhal's use of worker-to-worker communication in its common path also makes it more prone to be affected by AWS's various upload limitations, e.g., AWS upload bandwidth is half the stated download bandwidth, and there are network credit limits for "burst" uploading.

Line rate. Figure 8.9 illustrates Chop Chop's near line-rate network use by depicting its input, network and output rates:

- Input rate measures the total bytes of useful information—i.e., client identifiers and messages—that clients, load clients and load brokers all broadcast per time unit;
- Network rate measures the ingress bandwidth of servers at their network interface, i.e., useful information captured by the input rate as well as the Atomic Broadcast's overhead for ordering, authentication and deduplication;
- Output rate, or "goodput", measures the total bytes of useful information that each server delivers per time unit.

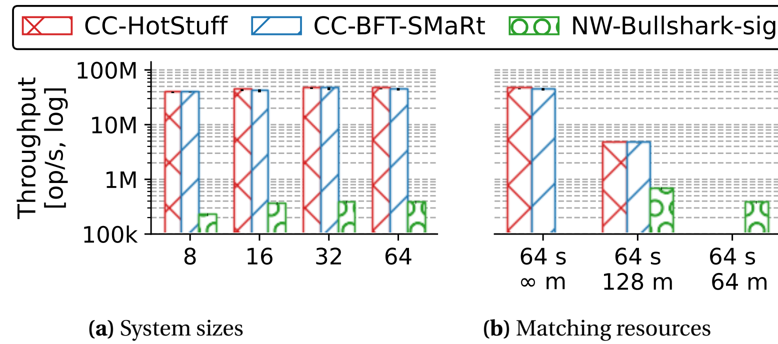
A system with perfect line rate would match all three rates: input rate would match output rate as messages can be delivered in a timely fashion with no backlogging, and output rate would match network rate as a server would only receive useful information, with no overhead due to Atomic Broadcast. The gray-shaded areas in Figure 8.9 highlight this overhead, i.e., the difference between network and input rates. Network and output rates are averaged over all servers.

In this experiment, each of the 257M simulated clients broadcast 8 B messages. This results in 11.5 B of useful information per broadcast as 28 bits = 3.5 B are sufficient to represent every identifier. This conversion is captured by the dotted line which converts the input rate from op/s, represented on the x-axis, to B/s, represented on the y-axis.

For authenticated Narwhal-Bullshark, the output rate closely matches the input rate until signature verification becomes the bottleneck at 378k op/s, shown by the plateauing output rate. The gap between Narwhal-Bullshark-sig's network and input rates is evident, differing by one order of magnitude (notably in line with our back-of-the-envelope calculation in Section 8.3.2). In contrast, thanks to distillation, Chop Chop practically achieves line-rate up to its maximum throughput. Before its inflection point at 40M op/s, the the overhead of Chop Chop is less than 8%. The drop in output and network rates at 60M op/s is due to servers surpassing their computational capacity: broadcasts stall, server witness verification gets backlogged and brokers, suspecting server faults, ask for more batch witnesses, further stressing servers' CPUs.

8.6.5 RQ3 – Number of Servers

Figure 8.10a illustrates the maximum throughput for systems of 8 ($f = 2$), 16 ($f = 5$), 32 ($f = 10$) and 64 ($f = 21$) servers. For Chop Chop, we adjust the witnessing margin as the system grows by 0, 1, 2, and 4 for 8, 16, 32 and 64 servers respectively (see Section 8.6.2). Both Chop Chop and authenticated Narwhal-Bullshark scale well to 64 servers. Note that, unless trust assumptions are modified, Narwhal-Bullshark-sig only scales vertically: if a Narwhal server or any of its workers are faulty, the entire server group is compromised. Chop Chop, instead, scales horizontally with the number of brokers.



(a) System sizes **(b)** Matching resources

Figure 8.10: Throughput of Chop Chop and authenticated Narwhal with Bullshark (log scale) when (a) varying system size, and when (b) varying the number of overall machines ("m") with 64 servers ("s"). Load brokers in Chop Chop simulate tens of brokers, hence are noted "∞ m".

8.6.6 RQ4 – Overall Efficiency

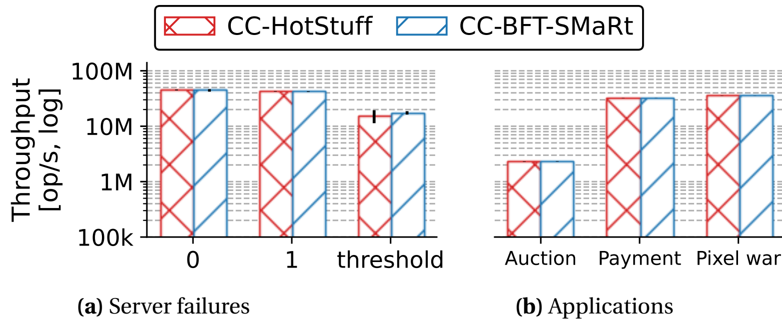
The center cluster of bars in Figure 8.10b compares Chop Chop's throughput with that of authenticated Narwhal-Bullshark when overall hardware resources are matched. In this setting, both systems have 128 machines at their disposal. Chop Chop is provided with 64 servers, 64 brokers and 0 load brokers. Since a load broker uses pre-generated synthetic data to simulate tens of brokers (see Section 8.6.2), involving load brokers in this experiment would give an unfair advantage to Chop Chop. Narwhal-Bullshark-sig is provided with 128 workers, to match Chop Chop's total machines, balanced across 64 server groups, to match Chop Chop's servers. The left and right clusters of bars depict Chop Chop using load brokers and Narwhal-Bullshark-sig with 64 server groups containing 1 worker each, respectively, as in the other experiments.

We observe 4.6M op/s for Chop Chop, with servers reporting around 5% CPU usage. We observe 679k op/s for Narwhal-Bullshark-sig. Chop Chop's higher throughput is in line with expectations. In Narwhal-Bullshark-sig, workers are trusted, and as such a worker can only contribute to its own server group. Instead, since Chop Chop brokers are untrusted, a broker's work is useful to all servers.

8.6.7 RQ5 – Chop Chop Under Failures

Figure 8.11a depicts Chop Chop's throughput when some servers crash 30 seconds into the run. Performance drops marginally (from 44M op/s to 43M op/s) with one crash and by 66% (down to 15M op/s) when one-third of the servers crash, resulting in less CPU globally available to witness batches.

Figure 8.8a captures Chop Chop's performance hit when clients fail to engage in distillation. This could be caused by clients being slow or crashed, or brokers being malicious. Under the most extreme conditions, where no client engages in distillation, the throughput drops from



(a) Server failures

(b) Applications

Figure 8.11: Throughput of Chop Chop (log scale) with (a) various server failures and for (b) different applications.

44M op/s to 1.5M op/s.

8.6.8 RQ6 – Application Use Cases

Figure 8.11b depicts the maximal stable throughput for various application use cases. In the Auction app, a client can bid an amount on a token it does not own, or take the highest offer it received for an item it owns. The highest amount bid on each token is locked and cannot be used to bid elsewhere. Money bid is transferred when the owner of the token takes the offer, or refunded when the bid is raised by another client. The Auction app is single-threaded and many clients bid on the same token to approximate a real auction. In the Payments app, clients choose a recipient and an amount to transfer. In Pixel war, clients choose a pixel and an RGB color to paint on a 2,048 by 2,048 board. Operations are generated at random.

We observe 2.3M op/s for the Auction, 32M op/s for Payments and 35M op/s for Pixel war. The bottleneck is the application in all cases, thus Chop Chop has sufficient capacity for high, single-application throughput. Chop Chop can also support many separate high-throughput applications simultaneously, making it a fitting Atomic Broadcast candidate to power a universal SMR system (or “Internet computer”).

8.7 Related Work

We overview below the state-of-the-art most relevant to Chop Chop, namely Atomic Broadcast systems with high-throughput and efficient signature aggregation schemes.

High-throughput Atomic Broadcast. Narwhal [55] is a mempool protocol that separates the reliable distribution of payloads from the communication-expensive ordering in order to accelerate DAG-based Atomic Broadcast [69, 92, 140]. Narwhal utilizes trusted workers to increase throughput while Chop Chop relies on *trustless* brokers, for the same effect, and scales out more efficiently. To circumvent the bottleneck associated with the broadcast leader,

approaches using multiple leaders have been developed—both for crash [64, 120] and arbitrary [7, 12, 141, 142] faults—to scale the broadcast throughput linearly with the number of leaders. Dissemination trees [96, 124] have also been employed to reduce communication cost and maximize network bandwidth utility, while sharded [98, 151] and federated [109] approaches reduce communication cost by promoting local communication in geo-distributed setups. In comparison, Chop Chop shows that an optimal distillation mechanism for batches achieves better performances without adding complexity to the Atomic Broadcast protocol itself.

Other approaches have shown that the underlying hardware of servers can also be exploited for higher throughput, such as FPGA [87, 91] and Intel SGX enclaves [16]. In comparison, Chop Chop uniquely boosts throughput by exploiting *trustless hardware* via brokers. Atomic Broadcast can also be accelerated in data centers by using the topology of the network [105, 127] or even by running within the network itself using P4-programmable switches [56, 97]. In such low latency environments, the processing overhead incurred by the operating system kernel can be bypassed to further increase the throughput of Atomic Broadcast [3, 97, 150].

Signature aggregation. Aggregate signatures were first proposed to save space by compacting a large number of signatures into just one [26, 136]. Up until recently, aggregation could also save verification time but only in certain cases: either when the signatures are generated by the same signer [42], or when the signatures are on the same message, i.e., multi-signatures [88]. In the latter case, aggregation mechanisms have been proposed to achieve constant-time verification of aggregated multi-signatures for both BLS [25] and Schnorr [115] signature schemes. In particular, multi-signatures are used in cryptocurrencies to have many servers sign the same batch of payloads [60, 96]. Servers in Chop Chop use rapidly-verifiable BLS multi-signatures [25] for that very purpose. In addition to aggregating server signatures on batches, Chop Chop's distillation mechanism also aggregates all client signatures in a batch in a way that provides constant-time verification. The theoretical scheme Draft [41] proposed signature aggregation with similar verification performances but is tailored to Reliable Broadcast. It is however unclear how Draft could be implemented as a real-world system without compromising liveness. Indeed, Draft assumes infinite memory to prevent message replay attacks which would rapidly exhaust servers' memory if Draft were to be deployed to match Chop Chop's target throughput in our evaluation (see Section 8.6.2). Chop Chop also aggregates client sequence numbers to significantly reduce bandwidth consumption when small messages are broadcast (Figure 8.2). Aggregating sequence numbers is made possible thanks to the ordering of Atomic Broadcast and thanks to novel legitimacy proofs (see Section 8.4.2).

Conclusions

This thesis focused on the scalability of Byzantine distributed computing, with the goal to develop systems that billions of stakeholders could partake in as servers, or use as clients. On the side of server scalability, we focused on Reliable Broadcast. We introduced Contagion, the first probabilistic Reliable Broadcast protocol to achieve logarithmic per-process computation and communication complexity. At the core of Contagion are samples, a novel alternative to quorums trading intersection guarantees for statistical representativeness. Contagion's analysis is, to our knowledge, the first to study a probabilistic distributed protocol in the Byzantine setting, enabled by a novel proving technique based on decorators. On the side of client scalability, we strove to maximize how efficiently Reliable Broadcast and Consensus servers make use of their resources when subject to a high throughput of requests. In the context of Reliable Broadcast, we introduced the notion of oracularity to describe a system's ability to match the performance of its centralized, trusted counterpart. We then introduced Draft, the first algorithm to achieve oracularity (at least in the good case), effectively achieving zero-cost Byzantine fault tolerance. To do so, we extended the distributed computing model with brokers, an untrusted layer of processes whose goal is to alleviate server complexity by performing heavy but verifiable operations. In the context of Consensus, we extended our broker-based techniques to the Atomic Broadcast abstraction. We introduced Chop Chop, the first near-line-rate system of its kind, processing tens of millions of operations per second on a geo-distributed deployment of medium-sized servers.

Our results, we believe, open interesting new avenues of future research. We outline a few of them below.

Scaling Consensus servers. As we discussed in the Introduction to this thesis, a great deal of effort was put towards generalizing our results on scalable Reliable Broadcast to the Consensus class. Doing so was challenging within the limited time frame of our PhD studies. While an algorithm, which we conjecture to be correct, was developed in the partially synchronous model, decorator-based proofs proved unwieldy when tackling the subtleties of probabilistic, adversarial time analysis. At the cost of additional formal effort, we believe a provably secure, logarithmic implementation of Atomic Broadcast should be attainable, completing the stated goal of this thesis.

Theory of decorators. We believe decorators, the tool we introduced to prove the correctness of Contagion, could be applied to the broader field of probabilistic Byzantine algorithms. To this end, a more general theory of decorators should be developed, streamlining our ad-hoc, decorator-based results into a usable toolbox of proving techniques. Significant (unpublished) progress was already made towards this goal, framing decorators in the broader context of probabilistic adversarial games.

Constant-memory Chop Chop. A global deployment of Chop Chop servers sustains a throughput of tens of millions of operations per second. In the real world, such a high throughput would realistically be generated by billions, maybe tens of billions of clients - even if each client persistently broadcast every two minutes, five billion clients would be necessary to incur a CPU bottleneck. Because each Chop Chop server must store the full list of all client public keys, such a large number of clients might severely strain server memory. More advanced cryptographic techniques, we believe, could enable offloading public key storage to brokers, effectively making Chop Chop a constant-memory algorithm. Doing so while maintaining near-line-rate performance presents a significant theoretical challenge.

Quantum resistance for reduction and distillation. The broker-based techniques we presented in this thesis to enhance batch authentication rely on multi-signature aggregation. State-of-the-art multi-signature schemes, such as BLS, generally rest their security on the hardness of the discrete logarithm problem, which is famously vulnerable to quantum attacks. While practical attacks are (to our knowledge) yet to be dealt, the general cryptographic consensus suggests pre-quantum schemes should be progressively phased out in the coming decade. Upgrading our systems with post-quantum authentication while preserving their communication complexity, we believe, will be of great importance in the near future.

Scaling out resource-intensive services. All applications showcased in this thesis (such as payments or auctions) have a minimal footprint in terms of computation and bit complexity. This choice was deliberate: our goal was to demonstrate our protocols would scale even in a setting where bottlenecks could not possibly be blamed on application complexity. Real-world applications, however, might be complex enough to severely bottleneck a fully-replicated system. This setting, we believe, calls for partial replication, with a core of secure, fully-replicated servers orchestrating computational resources contributed by the broader Internet. Doing so while maintaining (probabilistic) security would further enable the vision of a secure, planetary-scale computer shared by all.

Bibliography

[1] Ittai Abraham, Kartik Nayak, Ling Ren, and Zhuolun Xiang. 2021. Good-Case Latency of Byzantine Broadcast: A Complete Categorization. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[2] Daron Acemoglu and Asu Ozdaglar. 2009. Networks - Lecture 4: Erdős–Rényi Graphs and Phase Transitions. (2009).

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Antoine Murat, Athanasios Xygkis, and Igor Zablotchi. 2023. uBFT: Microsecond-Scale BFT Using Disaggregated Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[4] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. 2010. Reconfiguring Replicated Atomic Storage: A Tutorial. *Bulletin of the EATCS* (2010).

[5] Nicolas Alhaddad, Sisi Duan, Mayank Varia, and Haibin Zhang. 2021. Succinct Erasure Coding Proof Systems. *IACR Cryptology ePrint Archive* (2021).

[6] Dan Alistarh, Seth Gilbert, Rachid Guerraoui, and Morteza Zadimoghaddam. 2010. How Efficient Can Gossip Be? (On the Cost of Resilient Information Exchange). In *International Colloquium on Automata, Languages and Programming (ICALP)*.

[7] Salem Alqahtani and Murat Demirbas. 2021. BigBFT: A Multileader Byzantine Fault Tolerance Protocol for High Throughput. In *IEEE International Performance, Computing, and Communications Conference (IPCCC)*.

[8] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains. In *European Conference on Computer Systems (EuroSys)*.

[9] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *Journal of the ACM (JACM)* (1995).

[10] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. 2015. The Next 700 BFT Protocols. *ACM Transactions on Computer Systems (TOCS)* (2015).

[11] Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. 2021. Byzantine-Tolerant Causal Broadcast. *Theoretical Computer Science* (2021).

[12] Zeta Avarikioti, Lioba Heimbach, Roland Schmid, Laurent Vanbever, Roger Wattenhofer, and Patrick Wintermeyer. 2020. FnF-BFT: Exploring Performance Limits of BFT Protocols. In *International Colloquium on Structural Information and Communication Complexity (SIROCCO)*.

[13] Chen Avin, Michael Borokhovich, Keren Censor-Hillel, and Zvi Lotker. 2011. Order Optimal Information Spreading Using Algebraic Gossip. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[14] Baruch Awerbuch and Christian Scheideler. 2009. Towards a Scalable and Robust DHT. *Theory of Computing Systems* (2009).

[15] Paulo S. L. M. Barreto, Hae Y. Kim, Ben Lynn, and Michael Scott. 2002. Efficient Algorithms for Pairing-Based Cryptosystems. In *Annual International Cryptology Conference (CRYPTO)*.

[16] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *European Conference on Computer Systems (EuroSys)*.

[17] Michael Ben-Or, Ran Canetti, and Oded Goldreich. 1993. Asynchronous Secure Computation. In *Symposium on the Theory of Computing (STOC)*.

[18] Petra Berenbrink, Robert Elsaesser, and Tom Friedetzky. 2008. Efficient Randomised Broadcasting in Random Regular Networks With Applications in Peer-to-Peer Systems. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[19] Petra Berenbrink, Robert Elsässer, and Thomas Sauerwald. 2010. Communication Complexity of Quasirandom Rumor Spreading. In *European Symposium on Algorithms (ESA)*.

[20] Petra Berenbrink, Robert Elsässer, and Thomas Sauerwald. 2010. Randomised Broadcasting: Memory vs. Randomness. *Theoretical Computer Science* (2010).

[21] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman Speed Records. In *International Conference on Theory and Practice of Public Key Cryptography (PKC)*.

[22] Alysson Bessani, Joao Sousa, and Eduardo E.P. Alchieri. 2014. State Machine Replication for the Masses With BFT-SMART. In *Dependable Systems and Networks (DSN)*.

[23] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. 1999. Bimodal Multicast. *ACM Transactions on Computer Systems (TOCS)* (1999).

[24] Joseph T.A. Birman K.P. 1987. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems (TOCS)* (1987).

[25] Dan Boneh, Manu Drijvers, and Gregory Neven. 2018. Compact Multi-Signatures for Smaller Blockchains. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*.

[26] Dan Boneh, Craig Gentry, Ben Lynn, and Hovav Shacham. 2003. Aggregate and Verifiably Encrypted Signatures From Bilinear Maps. In *International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*.

[27] Dan Boneh, Sergey Gorbunov, Riad S. Wahby, Hoeteck Wee, and Zhenfei Zhang. 2022. BLS Signatures. RCF Draft.

[28] Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. 2009. Brahms: Byzantine Resilient Random Membership Sampling. *Computer Networks* (2009).

[29] Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. 2013. Communication Locality in Secure Multi-Party Computation. In *Theory of Cryptography Conference (TCC)*.

[30] Gabriel Bracha. 1984. An Asynchronous [(n-1)/3]-Resilient Consensus Protocol. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[31] Gabriel Bracha. 1987. Asynchronous Byzantine Agreement Protocols. *Information and Computation* (1987).

[32] Gabriel Bracha and Sam Toueg. 1985. Asynchronous Consensus and Broadcast Protocols. *Journal of the ACM (JACM)* (1985).

[33] Christian Cachin. 2010. State Machine Replication With Byzantine Faults. In *Replication*.

[34] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming*.

[35] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. In *Annual International Cryptology Conference (CRYPTO)*.

[36] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography. *Journal of Cryptology (JCrypt)* (2005).

[37] Christian Cachin and Jonathan A Poritz. 2002. Secure Intrusion-Tolerant Replication on the Internet. In *Dependable Systems and Networks (DSN)*.

[38] Christian Cachin and Stefano Tessaro. 2005. Asynchronous Verifiable Information Dispersal. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*.

[39] Martina Camaioni, Rachid Guerraoui, Jovan Komatovic, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, and Gauthier Voron. 2023. Carbon: Scaling Trusted Payments with Untrusted Machines. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2023). (Under review).

[40] Martina Camaioni, Rachid Guerraoui, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, and Gauthier Voron. 2024. Chop Chop: Byzantine Atomic Broadcast to the Network Limit. *USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2024). (Under minor revisions).

[41] Martina Camaioni, Rachid Guerraoui, Matteo Monti, and Manuel Vidigueira. 2022. Oracular Byzantine Reliable Broadcast. In *International Symposium on Distributed Computing (DISC)*.

[42] Jan Camenisch, Susan Hohenberger, and Michael Ostergaard Pedersen. 2012. Batch Verification of Short Signatures. *Journal of Cryptology (JCrypt)* (2012).

[43] Miguel Castro and Barbara Liskov. 1999. Practical Byzantine Fault Tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[44] Miguel Castro and Barbara Liskov. 2002. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems (TOCS)* (2002).

[45] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM (JACM)* (1996).

[46] Nishanth Chandran, Wutichai Chongchitmate, Juan A. Garay, Shafi Goldwasser, Rafail Ostrovsky, and Vassilis Zikas. 2015. The Hidden Graph Model: Communication Locality and Optimal Resiliency With Adaptive Faults. In *Innovations in Theoretical Computer Science (ITCS)*.

[47] Fan Chung and Linyuan Lu. 2001. The Diameter of Sparse Random Graphs. *Advances in Applied Mathematics* (2001).

[48] Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. 2023. On the Validity of Consensus. *International Symposium on Distributed Computing (DISC)* (2023).

[49] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. 2009. Upright Cluster Services. In *Symposium on Operating Systems Principles (SOSP)*.

[50] Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. 2020. Online Payments by Merely Broadcasting Messages. In *Dependable Systems and Networks (DSN)*.

[51] Thomas Cover and Joy Thomas. 2005. *Elements of Information Theory, Second Edition.*

[52] Tyler Crain, Christopher Natoli, and Vincent Gramoli. 2018. Evaluating the Red Belly Blockchain. *Computing Research Repository (CoRR)* (2018).

[53] Tyler Crain, Christopher Natoli, and Vincent Gramoli. 2021. Red Belly: A Secure, Fair and Scalable Open Blockchain. In *IEEE Symposium on Security and Privacy (S&P).*

[54] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability. In *IEEE Symposium on Security and Privacy (S&P).*

[55] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: A DAG-Based Mempool and Efficient BFT Consensus. In *European Conference on Computer Systems (EuroSys).*

[56] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. 2020. P4xos: Consensus as a Network Service. *IEEE/ACM Transactions on Networking (TON)* (2020).

[57] Sourav Das, Zhuolun Xiang, and Ling Ren. 2021. Asynchronous Data Dissemination and Its Applications. In *Annual ACM Conference on Computer and Communications Security (CCS).*

[58] Roger Dingledine, Nick Mathewson, and Paul Syverson. 2004. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium (SEC).*

[59] Danny Dolev and Rüdiger Reischuk. 1985. Bounds on Information Exchange for Byzantine Agreement. *Journal of the ACM (JACM)* (1985).

[60] Manu Drijvers, Sergey Gorbunov, Gregory Neven, and Hoeteck Wee. 2020. Pixel: Multi-Signatures for Consensus. In *USENIX Security Symposium (SEC).*

[61] Sisi Duan, Michael K. Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT Made Practical. In *Annual ACM Conference on Computer and Communications Security (CCS).*

[62] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys (CSUR)* (2004).

[63] Robert Elsässer and Dominik Kaaser. 2015. On the Influence of Graph Density on Randomized Gossiping. *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (2015).

[64] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. 2021. Efficient Replication via Timestamp Stability. In *European Conference on Computer Systems (EuroSys).*

[65] Paul Erdös and Alfréd Rényi. 1959. On Random Graphs. *Publicationes Mathematicae* (1959).

[66] P. Th. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. 2003. Lightweight Probabilistic Broadcast. *ACM Transactions on Computer Systems (TOCS)* (2003).

[67] Yaacov Fernandess, Antonio Fernández, and Maxime Monod. 2007. A Generic Theoretical Framework for Modeling Gossip-Based Algorithms. *ACM SIGOPS: Operating Systems Review* (2007).

[68] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus With One Faulty Process. *Journal of the ACM (JACM)* (1985).

[69] Adam Gagol, Damian Leundefinedniak, Damian Straszak, and Michal Swietek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks With Byzantine Nodes. In *Conference on Advances in Financial Technologies (AFT)*.

[70] Fangyu Gai, Jianyu Niu, Ivan Beschastnikh, Chen Feng, and Sheng Wang. 2023. Scaling Blockchain Consensus via a Robust Shared Mempool. In *IEEE International Conference on Data Engineering (ICDE)*.

[71] Juan Garay, Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. 2017. The Price of Low Communication in Secure Multi-Party Computation. In *Annual International Cryptology Conference (CRYPTO)*.

[72] Juan A Garay, Jonathan Katz, Ranjit Kumaresan, and Hong-Sheng Zhou. 2011. Adaptively Secure Broadcast, Revisited. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[73] Chryssis Georgiou, Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. 2008. On the Complexity of Asynchronous Gossip. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[74] Chryssis Georgiou, Seth Gilbert, Rachid Guerraoui, and Dariusz R. Kowalski. 2013. Asynchronous Gossip. *Journal of the ACM (JACM)* (2013).

[75] Chryssis Georgiou, Seth Gilbert, and Dariusz R. Kowalski. 2011. Meeting the Deadline: On the Complexity of Fault-Tolerant Continuous Gossip. *Distributed Computing* (2011).

[76] Mohsen Ghaffari and Merav Parter. 2016. A Polylogarithmic Gossip Algorithm for Plurality Consensus. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[77] George Giakkoupis, Yasamin Nazari, and Philipp Woelfel. 2016. How Asynchrony Affects Rumor Spreading Time. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[78] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. 2017. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Symposium on Operating Systems Principles (SOSP)*.

[79] Rachid Guerraoui, Florian Huc, and Anne-Marie Kermarrec. 2013. Highly Dynamic Distributed Computing With Byzantine Failures. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[80] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos Seredinschi. 2019. The Consensus Number of a Cryptocurrency. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[81] Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, and Dragos-Adrian Seredinschi. 2019. Scalable Byzantine Reliable Broadcast. *International Symposium on Distributed Computing (DISC)* (2019).

[82] Vassos Hadzilacos and Sam Toueg. 1993. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*.

[83] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical Accountability for Distributed Systems. In *Symposium on Operating Systems Principles (SOSP)*.

[84] Bernhard Haeupler, Gopal Pandurangan, David Peleg, Rajmohan Rajaraman, and Zhifeng Sun. 2012. Discovery Through Gossip. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*.

[85] James Hendricks, Gregory R Ganger, and Michael K Reiter. 2007. Verifying Distributed Erasure-Coded Data. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[86] Maurice Herlihy. 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1991).

[87] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. 2016. Consensus in a Box: Inexpensive Coordination in Hardware. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[88] Kazuharu Itakura and Katsuhiro Nakamura. 1983. A Public-Key Cryptosystem Suitable for Digital Multisignatures. *NEC Research & Development* (1983).

[89] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. 2009. T-Man: Gossip-Based Fast Overlay Topology Construction. *Computer Networks* (2009).

[90] Simon Josefsson and Ilari Liusvaara. 2017. Edwards-Curve Digital Signature Algorithm (EdDSA). RFC 8032.

[91] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. Cheap-BFT: Resource-Efficient Byzantine Fault Tolerance. In *European Conference on Computer Systems (EuroSys)*.

[92] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need Is DAG. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[93] Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. 2020. Order-Fairness for Byzantine Consensus. In *Annual International Cryptology Conference (CRYPTO)*.

[94] Valerie King, Steven Lonargan, Jared Saia, and Amitabh Trehan. 2011. Load Balanced Scalable Byzantine Agreement Through Quorum Building, With Full Information. In *International Conference of Distributed Computing and Networking (ICDCN)*.

[95] Valerie King, Jared Saia, Vishal Sanwalani, and Erik Vee. 2006. Scalable Leader Election. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*.

[96] Eleftherios Kokoris Kogias, Philipp Jovanovic, Nicolas Gailly, Ismail Khoffi, Linus Gasser, and Bryan Ford. 2016. Enhancing Bitcoin Security and Performance With Strong Consistency via Collective Signing. In *USENIX Security Symposium (SEC)*.

[97] Marios Kogias and Edouard Bugnion. 2020. HovercRaft: Achieving Scalability and Fault-Tolerance for Microsecond-Scale Datacenter Services. In *European Conference on Computer Systems (EuroSys)*.

[98] Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. 2018. OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding. In *IEEE Symposium on Security and Privacy (S&P)*.

[99] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. 2007. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Symposium on Operating Systems Principles (SOSP)*.

[100] Klaus Kursawe and Victor Shoup. 2005. Optimistic Asynchronous Atomic Broadcast. In *International Colloquium on Automata, Languages and Programming (ICALP)*.

[101] Petr Kuznetsov, Yvonne-Anne Pignolet, Pavel Ponomarev, and Andrei Tonkikh. 2021. Permissionless and Asynchronous Asset Transfer. In *International Symposium on Distributed Computing (DISC)*.

[102] Aptos Labs. 2022. The Aptos Blockchain: Safe, Scalable, and Upgradeable Web3 Infrastructure.

[103] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2010. Reconfiguring a State Machine. *SIGACT News* (2010).

[104] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1982).

[105] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus With Network Ordering. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[106] Meng-Jang Lin, Keith Marzullo, and Stefano Masini. 2000. Gossip Versus Deterministically Constrained Flooding on Small Networks. In *International Symposium on Distributed Computing (DISC)*.

[107] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. 2019. Teechain: A Secure Payment Network With Asynchronous Blockchain Access. In *Symposium on Operating Systems Principles (SOSP)*.

[108] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quema, and Marko Vukolic. 2016. XFT: Practical Fault Tolerance Beyond Crashes. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[109] Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. 2019. Fast and Secure Global Payments With Stellar. In *Symposium on Operating Systems Principles (SOSP)*.

[110] D. Malkhi, M. Merritt, and O. Rodeh. 1997. Secure Reliable Multicast Protocols in a WAN. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*.

[111] Dahlia Malkhi and Michael Reiter. 1996. A High-Throughput Secure Reliable Multicast Protocol. *IEEE Computer Security Foundations Symposium (CSF)* (1996).

[112] Dahlia Malkhi and Michael Reiter. 1997. Byzantine Quorum Systems. In *Symposium on the Theory of Computing (STOC)*.

[113] Dahlia Malkhi, Michael K Reiter, Avishai Wool, and Rebecca N Wright. 2001. Probabilistic Quorum Systems. *Information and Computation* (2001).

[114] Jean-Philippe Martin and Lorenzo Alvisi. 2005. Fast Byzantine Consensus. In *Dependable Systems and Networks (DSN)*.

[115] Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille. 2019. Simple Schnorr Multi-Signatures With Applications to Bitcoin. *Designs, Codes and Cryptography (DCC)* (2019).

[116] David Mazieres. 2016. The Stellar Consensus Protocol: A Federated Model for Internet-Level Consensus.

[117] Ralph C Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *Annual International Cryptology Conference (CRYPTO)*.

[118] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The Honey Badger of BFT Protocols. In *Annual ACM Conference on Computer and Communications Security (CCS)*.

[119] Peyman Momeni, Sergey Gorbunov, and Bohan Zhang. 2023. FairBlock: Preventing Blockchain Front-Running With Minimal Overheads. In *Security and Privacy in Communication Networks (SecureComm)*.

[120] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There Is More Consensus in Egalitarian Parliaments. In *Symposium on Operating Systems Principles (SOSP)*.

[121] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. 2014. Signature-Free Asynchronous Byzantine Consensus With $t < n/3$ and $O(n^2)$ Messages. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[122] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System.

[123] Kartik Nayak, Ling Ren, Elaine Shi, Nitin H Vaidya, and Zhuolun Xiang. 2020. Improved Extension Protocols for Byzantine Broadcast and Agreement. In *International Symposium on Distributed Computing (DISC)*.

[124] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. 2021. Kauri: Scalable BFT Consensus With Pipelined Tree-Based Dissemination and Aggregation. In *Symposium on Operating Systems Principles (SOSP)*.

[125] Fernando Pedone and André Schiper. 2002. Handling Message Semantics With Generic Broadcast Protocols. *Distributed Computing* (2002).

[126] James S Plank and Lihao Xu. 2006. Optimizing Cauchy Reed-Solomon Codes for Fault-Tolerant Network Storage Applications. In *IEEE International Symposium on Network Computing and Applications (NCA)*.

[127] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. 2015. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[128] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying Blockchain Extractable Value: How Dark Is the Forest?. In *IEEE Symposium on Security and Privacy (S&P)*.

[129] HariGovind V Ramasamy and Christian Cachin. 2005. Parsimonious Asynchronous Byzantine-Fault-Tolerant Atomic Broadcast. In *International Conference on Principles of Distributed Systems (OPODIS)*.

[130] Irving S Reed and Gustave Solomon. 1960. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics (SIAP)* (1960).

[131] Michael K. Reiter. 1994. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Annual ACM Conference on Computer and Communications Security (CCS)*.

[132] Michael K. Reiter and Kenneth P. Birman. 1994. How to Securely Replicate Services. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1994).

[133] Nuno Santos and André Schiper. 2013. Optimizing Paxos With Batching and Pipelining. *Theoretical Computer Science* (2013).

[134] Christian Scheideler. 2005. How to Spread Adversarial Nodes? Rotate!. In *Symposium on the Theory of Computing (STOC)*.

[135] Fred B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)* (1990).

[136] Claus P. Schnorr. 1991. Efficient Signature Generation by Smart Cards. *Journal of Cryptology (JCrypt)* (1991).

[137] Victor Shoup. 2000. Practical Threshold Signatures. In *International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*.

[138] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. 2008. BFT Protocols Under Fire. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[139] Suman Sourav, Peter Robinson, and Seth Gilbert. 2018. Slow Links, Fast Links, and the Cost of Gossip. *IEEE International Conference on Distributed Computing Systems (ICDCS)* (2018).

[140] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *Annual ACM Conference on Computer and Communications Security (CCS)*.

[141] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. 2022. State Machine Replication Scalability Made Simple. In *European Conference on Computer Systems (EuroSys)*.

[142] Chrysoula Stathakopoulou, David Tudor, Matej Pavlovic, and Marko Vukolić. 2022. [Solution] Mir-BFT: Scalable and Robust BFT for Decentralized Networks. *Journal of Systems Research (JSys)* (2022).

[143] The Diem Team. 2021. DiemBFT V4: State Machine Replication in the Diem Blockchain.

[144] The DFINITY Team. 2022. The Internet Computer for Geeks. *IACR Cryptology ePrint Archive* (2022).

[145] The MystenLabs Team. 2022. The Sui Smart Contracts Platform.

[146] Sam Toueg. 1984. Randomized Byzantine Agreements. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[147] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable Private Messaging Resistant to Traffic Analysis. In *Symposium on Operating Systems Principles (SOSP)*.

[148] Spyros Voulgaris, Márk Jelasity, and Maarten van Steen. 2004. A Robust and Scalable Peer-to-Peer Gossiping Protocol. In *Agents and Peer-to-Peer Computing (AP2PC)*.

[149] Marko Vukolic. 2010. The Origin of Quorum Systems. *Bulletin of the EATCS* (2010).

[150] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. 2017. APUS: Fast and Scalable Paxos on RDMA. In *ACM Symposium on Cloud Computing (SoCC)*.

[151] Jiaping Wang and Hao Wang. 2019. Monoxide: Scale Out Blockchain With Asynchronous Consensus Zones. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[152] Roger Wattenhofer. 2019. *Blockchain Science: Distributed Ledger Technology*.

[153] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger.

[154] Andrew Chi-Chih Yao. 1979. Some Complexity Questions Related to Distributive Computing. In *Symposium on the Theory of Computing (STOC)*.

[155] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. 2003. Separating Agreement From Execution for Byzantine Fault Tolerant Services. In *Symposium on Operating Systems Principles (SOSP)*.

[156] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus With Linearity and Responsiveness. In *ACM Symposium on Principles of Distributed Computing (PODC)*.

[157] B. Zhang, K. Han, B. Ravindran, and E. D. Jensen. 2008. RTQG: Real-Time Quorum-Based Gossip Protocol for Unreliable Networks. In *International Conference on Availability, Reliability and Security (ARES)*.

[158] Haoqian Zhang, Louis-Henri Merino, Vero Estrada-Galiñanes, and Bryan Ford. 2022. Flash Freezing Flash Boys: Countering Blockchain Front-Running. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*.

[159] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine Ordered Consensus Without Byzantine Oligarchy. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[160] Liyi Zhou, Kaihua Qin, Christof Ferreira Torres, Duc V Le, and Arthur Gervais. 2021. High-Frequency Trading on Decentralized on-Chain Exchanges. In *IEEE Symposium on Security and Privacy (S&P)*.

News Readings

[Cer19] Megan Cerullo. Google Reportedly Mining Millions of Americans Personal Health Data. *CBS News*, 2019.

[Con18] Nicholas Confessore. Cambridge Analytica and Facebook: The Scandal and the Fallout so Far. *The New York Times*, 2018.

[cyb23] Significant Cyber Incidents. *Center for Strategic & International Studies*, 2023.

[ESWV22] P. Ehin, M. Solvak, J. Willemson, and P. Vinkel. Internet Voting in Estonia 2005–2019: Evidence from Eleven Elections. *Government Information Quarterly*, 2022.

[Far22] Malcomb Farber. Cybercrime Damages to Cost the World $8 Trillion USD in 2023. *Associated Press*, 2022.

[Fit22] Afiq Fitri. Big Tech Now Accounts for More than Half of Global Internet Traffic. *Tech Monitor*, 2022.

[Gay21] Damien Gayle. Facebook Aware of Instagram's Harmful Effect on Teenage Girls, Leak Reveals. *The Guardian*, 2021.

[int21] IAB Study Shows Internet Economy is Transforming the U.S. Economy, Creating New Markets and Spurring Job Growth for Large and Small Businesses. *Interactive Advertising Bureau*, 2021.

[Moz18] Paul Mozur. A Genocide Incited on Facebook, with Posts from Myanmar's Military. *The New York Times*, 2018.

[Pri19] Rob Price. The FTC has Approved a Roughly $5 Billion Settlement with Facebook. *Business Insider*, 2019.

[Reu20] Reuters. Google Faces $5 Billion Lawsuit in U.S. for Tracking 'Private' Internet Use. *NBC News*, 2020.

[Run20] Dan Runkevicius. How Amazon Quietly Powers the Internet. *Forbes*, 2020.