Consensus in Blockchain: from Gossip to Synchronous Byzantine Fault Tolerance

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Nenad Milošević

under the supervision of
Fernando Pedone

September 2024

Dissertation Committee

Patrick Eugster	Università della Svizzera Italiana, Switzerland
Robert Soulé	Yale University, USA
Benoit Garbinato	University of Lausanne, Switzerland
Josef Widder	Informal Systems, Austria
Marko Vukolić	Protocol Labs

Dissertation accepted on 26 September 2024

Research Advisor	PhD Program Director
Fernando Pedone	**Walter Binder, Stefan Wolf**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Nenad Milošević
Lugano, 26 September 2024

To my beloved family

Abstract

State machine replication (SMR) and consensus are fundamental concepts in distributed systems, providing consistency, reliability, and fault tolerance. This thesis examines these problems in the context of modern decentralized systems, particularly blockchains. Specifically, it explores the integration of gossip communication with consensus protocols and investigates synchronous Byzantine fault-tolerant (BFT) consensus protocols.

We first explore the impact of gossip-based communication on consensus protocols, using the Paxos algorithm as a case study. We introduce Semantic Gossip, which optimizes gossip communication through semantic filtering and aggregation. Experimental results demonstrate that Semantic Gossip reduces message overhead and improves performance while maintaining reliability.

Next, we evaluate how synchrony violations impact the correctness of synchronous BFT consensus protocols, both with and without Byzantine attacks. We outline an experimental approach to determining a synchrony bound that, with high probability, prevents correctness violations. Applying this approach to a new protocol, BoundBFT, we find that communication diversity and redundancy enable BoundBFT to tolerate synchrony violations without compromising correctness, resulting in lower synchrony bounds and improved performance.

Finally, motivated by experimental data on message delays, we present a hybrid synchronous system model that distinguishes between small and large messages. Within this model, we develop AlterBFT, a BFT consensus protocol that relies on the timely delivery of small messages for agreement while requiring large messages to be eventually timely to ensure progress. Our evaluation shows that AlterBFT achieves significantly lower latency than state-of-the-art synchronous protocols and offers comparable performance and higher resilience than state-of-the-art partially synchronous protocols.

This thesis advances the understanding of consensus in partially connected and synchronous environments, providing practical solutions to improve the performance and fault tolerance of distributed systems.

Acknowledgements

First, I would like to express my deepest gratitude to my advisor, Professor Fernando Pedone, for his invaluable guidance, patience, and encouragement, which have profoundly shaped not only my research but also my perspective on academia and collaboration. His mentorship and unwavering trust were foundational to my growth, and every moment working together was a true pleasure and privilege. Fernando's example is one I deeply admire and will strive to emulate throughout my life.

I am also thankful to my thesis committee—Professors Benoit Garbinato, Patrick Eugster, Robert Soulé, Marko Vukolić, and Josef Widder—whose feedback and critical insights have strengthened my work. I am sincerely grateful for their expertise and commitment.

To my lab colleagues—Mojtaba, Elia, Lorenzo, Enrique, Tarcisio, and Long—thank you for the stimulating discussions, teamwork, and camaraderie that made our environment collaborative and motivating. A special thanks to Daniel Cason, whose close collaboration and support throughout my entire PhD were invaluable, especially in my early days.

I am also grateful to our Serbian community and friends in Ticino, whose warmth and support have truly made us feel at home here. To our wider family and dear friends back in Serbia, your support and love have been a constant source of strength and meaning throughout this journey.

Finally, I wish to thank my family. To my parents, Radinka and Milovan, for instilling true values in me and for their unwavering belief—you have been my foundation and inspiration. To my brother Zarko and sister Bojana, along with their families, thank you for your constant support and for being role models I admire deeply. Above all, to my beloved wife, Nevena—without your unwavering support, patience, and countless sacrifices, this journey would have been unimaginable. You are my greatest joy, and this accomplishment is as much yours as mine. To our wonderful sons, Aleksandar and Jovan, thank you for filling our life with boundless love, laughter, and purpose. You are my anchor, my greatest treasures, and my proudest achievements.

Preface

The result of this research appears in the following publications:

- D. Cason, N. Milosevic, Z. Milosevic, and F. Pedone. "Gossip Consensus". In Proceedings of the 22nd ACM/IFIP International Middleware Conference (Middleware 2021).
- N. Milosevic, D. Cason, Z. Milosevic, and F. Pedone. "How robust are synchronous consensus protocols?". In Proceedings of the 28th International Conference on Principles of Distributed Systems (OPODIS 2024).
- N. Milosevic, D. Cason, Z. Milosevic, and F. Pedone, "AlterBFT: Fast Synchronous BFT Consensus Protocol". Under submission.

Additionally, I was involved in the following publication, indirectly related to this thesis:

- D. Cason, Z. Milosevic, E. Fynn, E. Buchman, N. Milosevic, and F. Pedone. "The design, architecture and performance of the Tendermint Blockchain Network". In Proceedings of the 40th International Symposium on Reliable Distributed Systems (SRDS 2021).

Contents

List of Figures

List of Tables

Chapter 1

Introduction

State machine replication and consensus, as its core component, are crucial for the consistency, reliability, and fault tolerance of distributed systems. These problems have been extensively studied under various conditions, such as different synchrony assumptions, failure models, and roles assigned to processes. Most studies assume direct communication between processes in partially synchronous systems. Specifically, the network graph, where vertices represent processes executing consensus and edges represent the possibility of direct communication between two processes, is typically fully connected. The partially synchronous model enables the solution of the consensus problem that is otherwise unsolvable in a fully asynchronous system with failures, as stated by the FLP impossibility result [43]. In a partially synchronous system [37], the system is initially asynchronous but eventually becomes synchronous.

A new breed of decentralized systems, notably blockchain systems, has introduced a new environment. In decentralized systems, no single entity owns the infrastructure; instead, multiple entities from different administrative domains collaborate. In such environments, it is not always feasible for each process in one domain to communicate directly with all processes in another domain. For example, some processes may be behind firewalls, preventing direct connections to processes in other domains. Moreover, partially synchronous consensus protocols can tolerate fewer Byzantine processes compared to synchronous protocols. Synchronous consensus protocols can tolerate $f < n/2$ Byzantine or malicious processes out of n processes [42; 44; 64], an improvement over partially synchronous consensus protocols, which require $f < n/3$ [37]. This is particularly important for blockchain systems since higher fault tolerance enhances system security, as the adversary must possess more resources to harm the system.

Reaching consensus without full connectivity is challenging [6]. Some blockchain

consensus protocols address the partially connected network graph challenge by relying on gossip communication [9; 20; 21; 106]. Gossip protocols, which rely on rounds of message exchanges among neighboring processes, offer high communication reliability even in partially connected networks. However, it remains unclear to what extent consensus and gossip communication fit together. On the one hand, gossip communication has been shown to scale to large settings and efficiently handle participant failures and message losses. On the other hand, gossip may slow down consensus. Moreover, gossip's inherent redundancy may be unnecessary since consensus naturally accounts for participant failures and message losses.

As a result, the first research question we aim to answer in this thesis is: *Do consensus protocols and gossip-based communication indeed fit together?*

Synchronous Byzantine fault-tolerant (BFT) protocols have long been a reality in an academic setting, yet their practicality remains debated [3]. The primary concern is their dependency on a predefined synchronous bound, Δ, which is crucial for ensuring protocol correctness. This dependency creates a tradeoff between correctness and performance, as Δ directly impacts both. Moreover, most synchronous protocols require lock-step execution, where replicas must begin and complete each round together, hampering practical deployment due to potential delays and difficulties in maintaining perfect synchrony. Lastly, adversaries can exploit the dependence on synchrony through denial-of-service (DoS) attacks, hindering progress and potentially disrupting the system.

Modern synchronous protocols have successfully removed the impact of conservative time bounds on throughput and do not require lock-step execution [3; 58], boosting performance to levels comparable to partially synchronous protocols. Additionally, in blockchain applications, replicas are operated by independent entities, and there are strong system incentives for replicas to stay up and timely connected with the rest of the network.[1] Furthermore, replicas are often placed behind proxy nodes and thereby not directly exposed to DoS attacks.[2]

These improvements have opened up new perspectives for synchronous protocols. However, the community remains concerned that a single message exceeding the synchronous Δ bound could compromise protocol correctness. This leads us to our second research question: *How can we assess the impact of synchrony violations on the correctness of synchronous BFT consensus protocols, and*

[1] In most blockchain systems, protocol rewards depend on active and timely participation of nodes, and in some cases, there are built-in slashing mechanisms for nodes that miss consensus instances - https://tangem.com/en/blog/post/what-is-slashing/.

[2] Sentry node design for DDOS prevention - https://functionx.gitbook.io/home/fxcore-tutorials/sentry-nodes.

how can we further improve their performance?

1.1 Research contributions

In this section, we outline the main contributions of this dissertation and provide a short description of each one. We defer detailed discussions to the following chapters.

Gossip-based consensus communication (Chapter 3). We explore the deployment of consensus protocols in partially connected networks utilizing gossip communication. We use the Paxos consensus algorithm and investigate the impact of gossip-based communication on its performance, revealing significant latency and throughput overheads. To address these challenges, we introduce Semantic Gossip, a communication substrate optimized with two techniques: semantic filtering and semantic aggregation. Semantic filtering discards redundant messages based on consensus logic, while semantic aggregation combines multiple messages into a single message. Our experimental evaluation demonstrates that Semantic Gossip substantially reduces message overhead, improves performance, and maintains the reliability of gossip communication, even in the presence of message loss. These findings suggest that other agreement protocols could also benefit from similar optimizations.

Robustness of synchronous BFT consensus (Chapter 4). We propose a new approach to evaluating the resilience of synchronous BFT consensus protocols against synchrony violations, examining their behavior both in the presence and absence of Byzantine attacks. This approach is applied to BoundBFT, a novel synchronous BFT consensus protocol. Our experimental evaluation shows that, with communication diversity and redundancy, BoundBFT can tolerate synchrony violations while maintaining correctness. As a result, BoundBFT can operate with lower synchrony bounds and achieve enhanced performance compared to existing protocols, demonstrating the effectiveness of our proposed approach.

Hybrid synchronous system model and AlterBFT(Chapter 5). Based on the insights collected in a three-month experimental study on communication delays in a geographically distributed system, we propose a new hybrid synchronous system model. This model reflects our observed data better than the classical synchronous model. The new model distinguishes between small messages, which

adhere to strict timing bounds, and large messages that may exceed these bounds but are eventually delivered.

Assuming this model, we designed AlterBFT, a new BFT consensus protocol. AlterBFT relies on the timely delivery of small messages for agreement while allowing progress based on the eventual delivery of large messages. AlterBFT tolerates the same number of failures as synchronous protocols while achieving up to 15× lower latency and similar throughput. In addition to higher resilience, AlterBFT offers higher throughput and comparable latency to partially synchronous protocols.

1.2 Thesis outline

The rest of the thesis is organized as follows. Chapter 2 provides the foundations for the thesis, outlining the system model and definitions used throughout the text. Chapter 3 investigates the deployment and optimization of gossip-based communication for the Paxos consensus protocol, introducing Semantic Gossip to enhance its performance and maintain its reliability. Chapter 4 presents a new approach to assessing the resilience of synchronous BFT consensus protocols to synchrony violations and demonstrates its application through an evaluation of BoundBFT, a novel consensus protocol. Chapter 5 presents the hybrid synchronous system model and AlterBFT, the first BFT consensus protocol for the hybrid model. Chapter 6 surveys related work on the topics discussed in this thesis. Finally, Chapter 7 concludes the thesis by presenting our main findings and directions for future research.

Chapter 2

System model and definitions

In this chapter, we introduce the foundational concepts crucial for the thesis. We discuss the system models used, and define the consensus problem and blockchain.

2.1 System model

We consider a geographically distributed system consisting of a fixed set of processes, also called replicas. Processes communicate by exchanging messages without access to a shared memory or a global clock. Each process has its own local (hardware) clock, and while these clocks are not synchronized, they all run at the same speed [3]. We assume different failure models and timing models within this context.

2.1.1 Failure models

In this subsection, we explore the failure models considered in this thesis. We account for the possibility that a certain number of processes can be faulty and, depending on the types of failures, distinguish between two failure models. Processes that are not faulty are referred to as correct or honest.

Crash fault-tolerant model

In the crash fault-tolerant (CFT) model [57], a faulty process is one that eventually fails by crashing and, as a result, ceases to participate in the distributed algorithm without prior notice. Before crashing processes behave strictly according to the distributed algorithm.

Byzantine fault-tolerant model

In the Byzantine fault-tolerant (BFT) model [74], a faulty process can exhibit arbitrary (i.e., Byzantine) behavior, and adversaries can coordinate Byzantine processes.

We use digital signatures and a public-key infrastructure (PKI) to validate messages and detect Byzantine behavior (e.g., double-signing). A message m sent by process p is signed with p's private key and denoted as $\langle m \rangle_p$. Additionally, $id(v)$ represents the invocation of a random oracle that returns the unique hash of value v. Adversaries (and Byzantine processes under their control) are assumed computationally bound so that they are unable to subvert the cryptographic techniques used.

2.1.2 Timing models

Timing models define the assumptions about the bounds on relative process execution speeds and message transmission delays. The absence of such bounds characterizes an asynchronous system model, which does not permit solutions to important problems such as consensus [43]. Consequently, in this thesis, we consider models that assume some form of synchrony. Additionally, we assume that message transmission delays encompass execution time because in geographically distributed systems, process execution speeds are usually orders of magnitude smaller than message transmission delays.

Synchronous model

In the synchronous system model, there exists a known bound Δ on the maximal network transmission delay in communication between correct processes. We define a synchrony violation as a message taking more than Δ to be transmitted. We do not assume lock-step execution (e.g., [36; 74]); instead, we assume that all honest replicas start execution within Δ time [3; 5].[1] Processes communicate using point-to-point reliable links: every message an honest sender sends to an honest receiver is received.

[1] This can be implemented in a real system by having each replica broadcast a $Start$ message upon the beginning of the execution. A replica starts either upon receiving a $Start$ message from another replica or at a specific point in time.

Partially synchronous model

The partially synchronous system model [37] lies between a synchronous system and an asynchronous system. It relaxes the assumptions of the synchronous system by requiring that the timing bounds hold only eventually. We consider a variant of partial synchrony where the bounds are known but hold only after an unknown time called Global Stabilization Time (GST). Additionally, communication links between correct processes are reliable only after GST. Before GST, messages can be dropped, duplicated, or reordered, but we assume they cannot be corrupted.

2.2 Consensus

The consensus problem is one of the most fundamental problems in distributed computing [57]. It is most known for its role in state machine replication [99]. It defines a problem where a set of processes must reach a decision on a common value while a certain number of processes may be faulty. In this thesis, we consider the consensus problem in the context of both crash and Byzantine failures. The consensus definitions are slightly different in these two cases.

In the CFT model, consensus satisfies the following properties:

- *Uniform Agreement*: If a process decides on value v, then all correct processes eventually decide on v.
- *Uniform Integrity*: If a process decides on value v, then v was previously proposed by some process.
- *Progress*: Every correct process eventually decides on exactly one value.

In the BFT model, the consensus properties cannot be uniform because a Byzantine process can decide on an arbitrary value:

- *Agreement*: If a correct process decides on value v, then all correct processes eventually decide on v.
- *Integrity*: If all processes are correct and a process decides on value v, then v was previously proposed by some process.
- *Progress*: Every correct process eventually decides on exactly one value.

2.3 Blockchain

A blockchain is a distributed append-only log of transactions implemented by geographically distributed processes [91]. Unlike traditional consensus protocols that decide on a single value, a blockchain protocol forms a chain of values. A value in a blockchain is a block, and a block's position in the chain is referred to as the block's *height*. A block B_k at height k has the following format:

$$B_k \coloneqq (b_k, H(B_{k-1}))$$

where b_k denotes a proposed value (i.e., a set of transactions) and $H(B_{k-1})$ is a hash digest of the predecessor block. The first block, $B_1 = (b_1, \perp)$, has no predecessor. Every subsequent block B_k must specify a predecessor block B_{k-1} by including a hash of it.

A block is considered valid if (i) its predecessor is valid or $\perp$, and (ii) its proposed value meets application-level validity conditions and is consistent with its chain of ancestors (e.g., there are no double-spending transactions). If block B_k is an ancestor of block B_l (i.e., $l \geq k$), we say B_l *extends* B_k. We say blocks B_l and $B'_{l'}$ *equivocate* each other if they do not extend one another.

We assume that a blockchain (consensus) protocol must satisfy the following properties [3]:

- *Agreement*: No two honest replicas commit different blocks at the same height.
- *Progress*: All honest replicas keep committing new blocks.
- *External validity*: Every committed block satisfies the predefined *valid()* predicate.

This variant of the consensus problem has an application-specific $valid()$ predicate to indicate whether a block is valid [20; 23]. For instance, in blockchain systems, a block is considered invalid if it lacks the correct hash of the preceding block.

Chapter 3

Gossip Consensus

3.1 Introduction

Gossip-based consensus protocols have been proposed to address the challenges faced by state machine replication in large, geographically distributed systems. In this chapter, we investigate the suitability of gossip as a communication building block for consensus. We aim to answer three questions: How much overhead does classic gossip introduce in consensus? Can we design consensus-friendly gossip protocols? Would efficient gossip protocols maintain the same reliability properties as classic gossip?

We examine gossip-based consensus from a systems perspective. Since gossip provides high communication reliability and some consensus algorithms can handle message losses (e.g., Paxos [70]), one could naturally layer a consensus protocol on top of a gossip communication protocol (see Figure 3.1) without designing a consensus protocol from scratch for a partially connected network graph. Consequently, we consider a particular consensus protocol, Paxos [70], and experimentally study its behavior when relying on gossip communication (Section 3.4).

Our choice of Paxos is justified as follows: (a) Paxos is sufficiently known in the distributed systems community and needs no lengthy explanation; (b) while Paxos is not simple, it is simpler than many other consensus protocols (e.g., [24; 72; 75; 90]); (c) process interactions in Paxos include all communication patterns of interest (i.e., one-to-one, one-to-many, many-to-one, and many-to-many), which renders our study of general interest; and (d) Paxos is a viable option for decentralized systems that tolerate crash failures only (e.g., [9]).

We start by considering the impact of gossip communication on the performance of consensus (i.e., throughput and latency). Unsurprisingly, Paxos atop

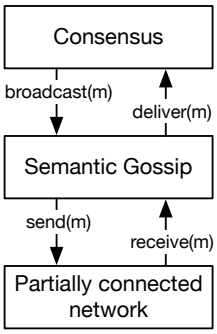

Figure 3.1. Gossip-based consensus.

gossip performs poorly in terms of throughput and latency when compared to a baseline Paxos deployment where processes can communicate directly with the Paxos coordinator.

The adoption of gossip causes a latency degradation of up to 52% in our experiments, while the maximum achieved throughput can be reduced by up to 74%, depending on the system size. Although the comparison is not fair, as Paxos atop gossip can run in a partially connected network, while baseline Paxos assumes a fully connected network, it provides a reference.

We then consider designing and implementing a "consensus-friendly" gossip communication substrate. The idea is to reduce the overhead of gossip by exploiting consensus semantics. We introduce Semantic Gossip, which optimizes classic gossip with two techniques, *semantic filtering* and *semantic aggregation*. Semantic filtering allows the gossip layer to discard messages that have become dispensable, according to the consensus logic. In the case of Paxos, decision messages render voting messages irrelevant. Thus, once a process starts propagating a decision message, it stops the propagation of any voting messages that lead to the decision. Semantic aggregation allows processes to group multiple messages into a single one with equivalent meaning to consensus. In Paxos, multiple voting messages can be grouped into a single multi-process voting message. Notice that while Semantic Gossip uses knowledge about Paxos, no changes are required in the implementation of Paxos.

Semantic filtering and aggregation substantially reduce the number of messages exchanged by processes to reach consensus. When both techniques are combined, the reduction can be up to 58%, compared to messages exchanged with classic gossip.

Moreover, Semantic Gossip boosts the performance of Paxos when compared to implementations based on classic gossip. The adoption of the two semantic techniques improves the latency of gossip-based Paxos by from 7% to 24%, while

it is also able to sustain higher workloads than the Paxos implementation based on classic gossip.

The performance improvements brought by Semantic Gossip are welcome as long as they do not come at the expense of the reliability that classic gossip provides. To consider this aspect, we inject failures (i.e., message loss) in the execution of Paxos based on both classic gossip and Semantic Gossip. We found that Semantic Gossip-based Paxos retains the resilience of gossip, up to 20% of injected message loss.

3.1.1 Outline

The remainder of this chapter is organized as follows. Section 3.2 details the system model and introduces background information on gossip and Paxos. Section 3.3 proposes the design and implementation of Semantic Gossip. Section 3.4 describes the evaluation of Paxos using point-to-point, classic gossip, and Semantic Gossip communication. Section 3.5 concludes the chapter.

3.2 Background

In this section, we provide some basic background on gossip communication and the adopted consensus algorithm (Paxos [70]).

3.2.1 Gossip communication

The gossip communication approach is derived from epidemic dissemination strategies used to propagate information in a distributed system. Originally proposed for the dissemination of updates in replicated databases [35], epidemic algorithms have proven to be an efficient and resilient approach to implementing multicast and broadcast primitives [15]. The operation of epidemic dissemination consists of periodic message-exchange rounds, in which every process randomly selects other processes with which to interact.

There are three general gossip dissemination strategies. In the *push* strategy, every process that has updates (i.e., new messages) to propagate sends them to the selected peers. In the *pull* strategy, processes request updates to the selected peers, which transmit the updates, if they have any, to the requesting processes. These two strategies can be combined into a *push-pull* strategy, in which processes in a round can both send updates to peers and receive updates from them.

The *push*, *pull*, and *push-pull* strategies differ in terms of performance, the number of messages exchanged, and the number of rounds to contact a given portion of the processes with high probability. The best strategy typically depends on the application behavior, the size and frequency of updates, and on the methods used to control the dissemination [35]. In this work we adopt the *push* strategy, however, our contributions could be extended to other strategies.

An algorithm interacts with the gossip communication layer using a *broadcast* primitive that addresses a message to all processes. It is a non-blocking primitive, as the dissemination is asynchronous and may take several rounds. The *deliver* primitive returns messages broadcast by processes. It is a blocking primitive returning messages locally broadcast and messages received from other processes. There are no guarantees that a message broadcast by a non-faulty process is delivered by all non-faulty processes; due to process or link failures, a message may never reach some destinations. In addition, the random choice of peers to which messages are sent may not provide full connectivity. However, a proper choice of parameters (i.e., number of rounds and the number of processes to which a processes gossips in each round) provides very high reliability, specially when the *push* dissemination strategy is adopted [15].

3.2.2 Paxos

Paxos [70] is an algorithm that implements consensus (Section 2.2) with crash failures (Section 2.1.1) in a partially synchronous system (Section 2.1.2). It can be used to implement state machine replication [99] by running multiple independent instances of consensus, each identified by a positive integer, where each instance decides on a single value. The output of the algorithm consists of the values decided in subsequent instances of consensus, following the total order established by instance identifiers, with no gaps.

Paxos distinguishes among the roles that processes play in the execution of the algorithm: proposers, acceptors, and learners. We assume that each Paxos process plays all these roles. Thus, a process proposes values, works to ensure that a single value is accepted in each instance of consensus, and learns the decided values. Paxos has been optimized in many ways (e.g., [13; 72; 83; 85; 90; 94]). We adopt the classic version of the algorithm, described in [70].

Each instance of consensus proceeds in rounds, identified by positive integers. Each round is orchestrated by a process, the coordinator. A coordinator can start the same round in multiple instances of consensus. A round consists of two phases, Phase 1 and Phase 2. In each phase, the coordinator sends a message, either a Phase 1a or Phase 2a message, to all processes (one-to-many

communication pattern) and waits for Phase 1b or Phase 2b reply messages from a majority of them (many-to-one communication pattern). Messages are tagged with the identifier of the instance they belong to. A process replies to the coordinator of a round provided that it has not replied to messages from higher-numbered rounds in that instance of consensus. In Phase 1, the coordinator of a round tries to find out if a value may have been chosen in lower-numbered rounds. In Phase 2, the coordinator asks the processes to accept a value, either learned from Phase 1b messages or any value proposed by a client.

When a majority of processes accept a value in the Phase 2 of a round, the value of that instance of consensus is decided. Paxos ensures that no other values can be chosen in higher-numbered rounds of that instance of consensus, as at least one process will report the accepted value in Phase 1. When the coordinator learns, from Phase 2b messages, that a value is decided, it informs all processes using a Decision message (one-to-many communication pattern). This communication step becomes redundant if Phase 2b messages are received by all non-faulty processes, not only by the coordinator.

Paxos is safe in the presence of concurrent coordinators, but for the sake of progress a single process is expected to act as the coordinator at a time. Once elected as the coordinator, a process starts a round in multiple instances of consensus at once. In a restricted set of instances processes may have accepted values, forcing the coordinator to re-propose them in Phase 2. But for most instances no process will report having accepted values in previous rounds; in these cases, the coordinator is free to propose any value in Phase 2. Thus, in regular (fail-free) operation, the decision of a value only requires the execution of Phase 2 of a round [70].

3.3 Semantic Gossip

In this section, we motivate the need for gossip protocols optimized for consensus, describe the design of a gossip protocol that takes advantage of consensus semantics, and detail its implementation.

3.3.1 Motivation

Implementing consensus on top of gossip communication is straightforward. Essentially, the original communication layer, which assumes a fully connected network graph and provides (one-to-one) *send* and *receive* primitives, is replaced by

a partially connected network with a gossip communication layer, which provides (one-to-many) *broadcast* and *deliver* primitives (Figure 3.2).

In Paxos, Phase 1a and Phase 2a messages, sent by the coordinator to all processes (one-to-many communication pattern) naturally benefit from gossip communication. Instead of sending Phase 1a and Phase 2a messages to all processes the coordinator is directly connected to, it can broadcast the messages via gossip. Eventually, with a reasonably high probability, the messages are delivered to all Paxos participants and their propagation cease. Notice that Paxos tolerates message loss, so probabilistic delivery guarantees are sufficient.

Gossip is not well-suited for propagating Phase 1b messages, from all processes to the coordinator (many-to-one communication pattern), as these messages only concern the coordinator, but will be delivered to all participants. Fortunately, Phase 1b messages are rarely sent during regular, fail-free operation; thus, the overhead of propagating them via gossip should not have relevant impact on the overall performance. In the case of Phase 2b messages, the fact that they will be delivered to all processes, not only to the coordinator, is positive. In fact, processes do not need to wait for a Decision message from the coordinator if they receive identical Phase 2b messages from a majority of processes. As a result, the propagation of Phase 2b messages via gossip may speed up decisions.

The mismatch between Paxos, and more generally a fault-tolerant consensus protocol, and gossip communication stems from the fact that Paxos was designed to tolerate process crashes and message losses, while gossip protocols strive to provide probabilistic reliable communication. Both consensus and gossip achieve their guarantees by means of communication redundancy. The result is an unnecessarily high number of message exchanges, which penalizes performance. The degree of redundancy increases when using gossip communication, as processes are likely to receive the same message multiple times, from different peers.

The use of gossip as an underlying means of communication, however, is beneficial for Paxos since gossip does not require direct communication between every pair of processes. This feature naturally extends to environments in which processes are connected to subsets of processes only, and balances the communication load among processes.

3.3.2 Design

In this section, we discuss simple techniques to address the mismatch between a fault-tolerant consensus protocol, using Paxos as a reference, and the underlying gossip communication substrate. The goal is essentially to reduce the message redundancy at the gossip layer, employing the knowledge about the message se-

mantics provided by the consensus protocol. The challenge is to achieve this reduction in message redundancy without sacrificing modularity (i.e., without modifying the original Paxos protocol) and the original resilience guarantees offered by gossip.

Semantic filtering. The first technique provides to the consensus protocol the ability to decide whether a message should be sent to a peer. This means interfering with the operation of the gossip layer, which by default forwards every message to all peers. The consensus protocol can then restrain the propagation of messages that are (potentially) no longer useful to a peer, and therefore to all other processes the peer is connected to. The main goal is to save network and processing resources that would be used to forward messages that peers will probably disregard.

Semantic filtering is implemented through a set of rules to identify messages that, according to the consensus semantics, have become obsolete or redundant. For instance, a message from a given round of consensus typically renders any message from previous (smaller) rounds obsolete. Or, for some round steps, acknowledgements from a majority of processes may render further acknowledgements redundant. The semantic filtering rules are evaluated when a message is ready to be sent to a peer. If the message is filtered out, because it is identified as either obsolete or redundant, the gossip layer discards it; otherwise, it is sent as usual.

The evaluation of the semantic filtering rules can be seen as a lightweight execution of the consensus protocol on behalf of a peer. In fact, to identify messages that can be filtered out it is necessary to store some information about messages that were previously sent to that peer. The more comprehensive the rules are, the more information is stored per peer, and the more costly is to evaluate them. Thus, the choice of a set of semantic filtering rules should balance the cost of evaluating them for every message forwarded, with the benefits that an effective filtering can provide.

In the case of Paxos, the proposed semantic filtering rules affect the propagation of Decision and Phase 2b messages. A Decision message is broadcast by the coordinator when it receives Phase 2b messages from a given round and instance from a majority of processes. A Decision message from a given instance thus renders any Phase 2b message from that same instance obsolete. A process can also learn the value decided in an instance of consensus by receiving identical Phase 2b messages from a majority of processes. From this point on, any further Phase 2b message from the same instance becomes redundant. In both cases, Phase 2b

messages are not forwarded to a peer when they refer to an instance for which the peer is expected to already know the decision, from the messages previously sent to it.

Semantic aggregation. The second technique provides the consensus protocol with the possibility to replace a number of similar or related messages, which will be sent to a peer, with a single message comprising the information carried by the original messages. This technique explores scenarios in which the gossip layer has multiple pending messages to send to a peer and some of them are likely to be aggregated, according to the consensus semantics. It is an opportunistic mechanism that aims to reduce the number of messages exchanged by processes via gossip, especially when they operate under moderate to high load.

Semantic aggregation is also implemented through a set of rules that, from a list of pending messages: (i) identify those that are prone to aggregation, and (ii) define how an aggregated message can be built from the original messages. When messages prone to aggregation are found, the first of them in the list of pending messages is replaced by the aggregated message, built according to the respective rule, while the remaining ones are removed from the list. In other words, an aggregated message both replaces and filters out the original messages that it aggregates. Messages that are not prone to aggregation, or for which aggregation is not deemed advantageous by the consensus protocol, are not affected by this technique. They are kept in the list of pending messages, and are forwarded to the peers as usual.

Semantic aggregation rules can be either reversible or not. When a process receives from a peer a message aggregated using a reversible rule, it reconstructs the original messages and treats them as regular messages. That is, messages that are received for the first time are delivered to the consensus protocol and forwarded to other peers—in this process, in particular, they can be semantically aggregated again. When an aggregated message is built from a non-reversible rule, it is treated as a new message broadcast by the process that aggregated it. In this case, the consensus protocol must be able to handle the semantically aggregated message.

Observe that, despite the similarities, semantic aggregation is not the same as batching [45]. When implemented at the network level, batching essentially concatenates messages, treated as raw byte arrays, to optimize the network usage. At the application level, some message types are batched until the batch size reaches a threshold or a timeout expires. As a result, batching can have a negative effect on performance when the system is subject to low loads, as the sending

of messages is delayed. This does not happen with semantic aggregation, which despite being ineffective under low loads, does not delay the sending of any messages. Moreover, the technique is more flexible than batching, as messages are not only concatenated, but can be transformed or merged in any arbitrary ways defined by the semantic aggregation rules. So, while the size of a batch of messages is proportional to the number of messages in the batch, an aggregated vote message, for instance, has essentially the same size regardless of the number of single vote messages it has replaced.

As for semantic filtering, the best candidates in Paxos for semantic aggregation are Phase 2b messages. When there are multiple identical Phase 2b messages pending to be sent to a process, they can be easily replaced by a single message. For this, a single semantic aggregation rule was adopted. It considers for aggregation Phase 2b messages referring to the same instance and round of consensus; so they only differ by their senders. The aggregated message consists of any of the original Phase 2b messages plus a field to store the multiple senders. As reconstructing the original Phase 2b is straightforward, the aggregation rule is reversible and no changes in the Paxos protocol were required.

3.3.3 Implementation

We implemented a gossip-based communication layer to interconnect processes. At the system setup, each process opens connections to a randomly selected set of k processes, where k is a system parameter. Connections are bi-directional, so that the set of peers of a process includes both the k peers to which it opened connections, and a number of peers from which it received connection requests. In fact, the expected number of peers each process interacts with is $2k$.

Classic gossip. Figure 3.2 illustrates the architecture of the gossip layer. A process interacts with the consensus protocol via two queues. The *broadcast queue* is fed by locally broadcast messages, and the *delivery queue* offers messages to the consensus protocol. A process also maintains, for each peer it is connected to, a Send and a Receive routine. A *send queue* is associated to each Send routine; messages added to a *send queue* are eventually sent to the corresponding peer. There is a single *receive queue* shared by all Receive routines, to which messages received from all active peers are added. A message added to the *broadcast queue* is locally delivered and sent to all peers: it is added to the *delivery queue* and to all active *send queues*. A message added to the *receive queue* is delivered and forwarded to all peers but the peer the message came from: it is added to the *delivery queue* and to all, but the message's origin, *send queues*. The selection of

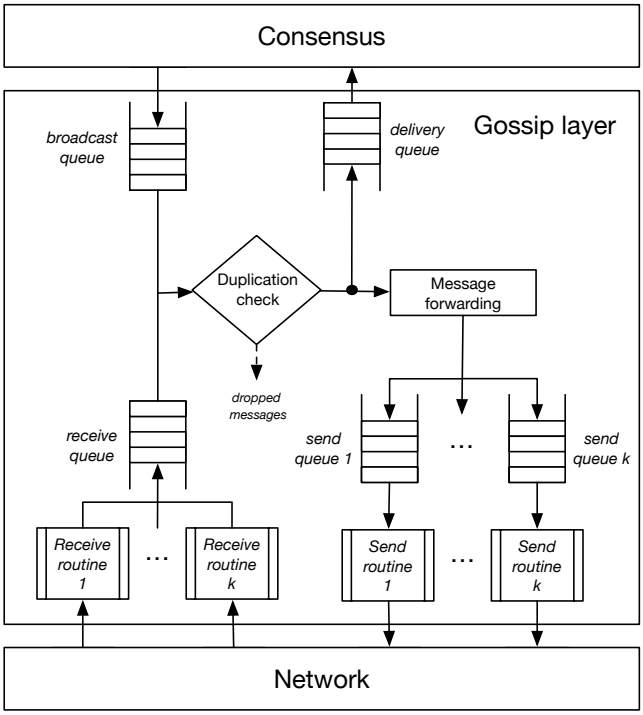

Figure 3.2. Architecture of the gossip layer at a process.

peers to which a message is sent is done by the Message forwarding module from the gossip main routine.

Messages are propagated using the *push* disseminating strategy. This means that the same message can be received by a process several times, from distinct peers. We control the flooding of messages using a simple approach based on a cache of *recently seen* messages, maintained by every process. A message is registered to the *recently seen* cache before it is delivered to the consensus protocol and sent to the process' peers. If the same message is received within a short period of time, so that the message's identifier is still on the *recently seen* cache, the message is dropped—i.e., it is not delivered nor forwarded to the peers. This is the role of the Duplication check module represented in Figure 3.2: it prevents, with some probability, a message from being delivered and forwarded more than once. There is no actual guarantee of a deliver-and-forward once behavior, but the adoption of a reasonable *recently seen* cache size (e.g., 256 KB) reduces the probability of message duplication. It is worth noting that the *recently seen* cache stores message unique identifiers, that can be defined by the consensus protocol to prevent hash collisions, and not full messages, so its size is constant and relatively small. The same functionality could be obtained by adopting other

approaches, such as a sliding Bloom filter [92].

Semantic extensions. The gossip layer offers two ways to control its behavior, the techniques presented in Section 3.3.2: semantic filtering and semantic aggregation. The consensus protocol can adopt one or both techniques by implementing interface methods offered by the gossip layer.

Semantic filtering is provided by allowing the consensus protocol to implement a *validate* method, which receives a message and a destination peer, and returns a boolean:

```
Bool validate(Message, Peer)
```

The *validate* method is invoked by a Send routine when it is ready to send a message to the respective peer. If the method returns false, the message is dropped, as the decision was to filter out the message. Otherwise, the message is sent to the peer, the default behavior when the method is not implemented. Implementations of the *validate* method should be fast and non-blocking, as it is likely to be invoked concurrently by multiple sending routines. The implementation should keep some information about the state of each peer, essentially a summary of relevant messages that were previously processed and not filtered out, and thus sent to that peer. The cost of storing such information versus the benefit in terms of resource saving by filtering out messages that would be sent to a peer should be considered.

Semantic aggregation is provided through the implementation of a pair of methods, *aggregate* and *disaggregate*:

```
Message[] aggregate(Message[], Peer)
Message[] disaggregate(Message)
```

The *aggregate* method receives an array of messages and a destination peer, and returns an array of messages. It is invoked by a Send routine when it has multiple pending messages to be sent to the respective peer. Messages returned by the *aggregate* method, both original and aggregated ones, are sent to the peer, in the order in which they are returned. The *disaggregate* method receives an aggregated message and returns either an array of reconstructed messages, for reversible semantically aggregated messages, or the same message received otherwise. It is invoked by the main gossip routine of a process when a message marked as aggregated is received from a peer. Messages returned by the method are processed as regular messages, in the order in which they are returned: they are checked against the *recently seen* cache and, if not duplicated, delivered and forwarded to peers.

3.4 Experimental evaluation

In this section, we first introduce the evaluation setup (Section 3.4.1). Then, we evaluate the effect of applying semantic extensions on the performance (Section 3.4.2) and reliability (Section 3.4.3) of Paxos algorithm. Next, we discuss the results in light of different network topologies (Section 3.4.4), and conclude with a summary of the main findings (Section 3.4.5).

3.4.1 Evaluation setup

This section explains the evaluation methodology, followed by details about the experimental environment and implementation.

Methodology

We carried out experiments with Paxos using three setups, which differ by the implementation of the communication substrate, while sharing the same Paxos implementation.

The first setup, *Baseline*, provides a reference for the performance of a classic Paxos deployment (with three phases, as presented in Section 3.2.2) in the experimental environment. In the Baseline setup, the Paxos coordinator communicates directly with every other process, which essentially assumes a fully connected network. Nevertheless, comparing to the Baseline setup illustrates the inherent difficulty of designing protocols for multi-administrative environments: performance-wise and under normal conditions, the Baseline setup provides a best case.

In the second setup, *Gossip*, processes can communicate directly only with a small subset of other processes. Communication takes place via gosssip, and messages are disseminated through a randomly generated overlay network. In the third setup, *Semantic Gossip*, processes communicate in the same way as in the Gossip setup, but the gossip layer is augmented with the semantic filtering and aggregation techniques described in Section 3.3.2.

Environment

We conducted the main experiments in a geographically distributed environment, with processes evenly spread among 13 AWS regions: North Virginia, Canada, Northern California, Oregon, London, Ireland, Frankfurt, São Paulo,

Region	Latency (ms)
Canada	7
N. California	30
Oregon	39
London	38
Ireland	33
Frankfurt	44
S.Paulo	58
Tokyo	73
Mumbai	93
Sydney	98
Seoul	87
Singapore	105

Table 3.1. WAN latencies between the coordinator's region (North Virginia) and the other twelve regions.

Tokyo, Mumbai, Sydney, Seoul, and Singapore. We have placed the Paxos coordinator in North Virginia in all experiments because it is the region that has the lowest latency to and from all other regions.

Table 3.1 lists the WAN latencies between the coordinator's region (North Virginia) and the other twelve regions. Processes were hosted by t2.medium Amazon EC2 instances, with 2 vCPUs and 4GB of RAM.

We conducted an additional set of experiments in a cluster, where we emulated the wide-area latencies between the above mentioned 13 AWS zones. Latencies between cluster nodes were configured using the Linux Traffic Control kernel module [62], that allows postponing the sending of messages to a given destination for a provided delay. The emulated WAN provided an affordable approximation of the AWS environment for experiments requiring hundreds of executions. Those experiments we carried out in a cluster with two groups of machines: (i) Dell PowerEdge 1435 with two Dual-Core AMD Opteron 2GHz and 4GB of RAM, and (ii) HP SE1102 with two Quad-Core Intel Xeon 2.5GHz and 8GB of RAM. By hosting two processes in nodes of group (ii), the performance observed in the emulated environment was comparable with the performance observed in AWS.

Implementation

We implemented Paxos, the gossip communication layer, and the Semantic Gossip extensions in Go. We rely on libp2p [78] to establish and maintain communication channels between pairs of processes. Libp2p channels are built atop TCP connections, and provide encryption, multiplexing, flow control, and network-level batching. Although libp2p channels are reliable, our implementation may

discard messages when queues connecting different routines are full, as a way to prevent slow processes from blocking the main transport routine. In addition, libp2p connections may be dropped when receivers are much slower than senders; although the dropped connections are reestablished, some messages may be lost. Temporary disconnections between peers, however, do not compromise the network connectivity.

The same Paxos implementation was used for all setups. In the Baseline setup, the elected Paxos coordinator opens libp2p channels to all other processes, which during fail-free operation only interact with the coordinator. In the Gossip and Semantic Gossip setups, each process opens a libp2p channel to a random subset of k processes. We set k to $log_2 n$, where n is the system size. As a result, each process communicates directly with $2log_2 n$ other processes on average. This number of connections per process ensures, with high probability, that the generated network overlay is connected [39]. To provide a fair comparison of results, for each system size n, we enforce the same network overlay in experiments with Gossip and Semantic Gossip setups. In Section 3.4.4, we then consider multiple randomly generated network overlays and argue that this choice does not affect our conclusions.

Clients generate an experiment workload by proposing values to Paxos. There is one client per region that submits values to a Paxos process hosted in the same region as the client. The communication between clients and Paxos processes is reliable. When a Paxos process receives a value from a client, it forwards the value to the coordinator; the coordinator then proposes the client value in Phase 2 of the next unused Paxos instance. Paxos processes inform all connected clients about Paxos decisions. When a client is notified of the decision of a value it has submitted, it computes the end-to-end latency; throughput is computed as the rate of decisions per time unit. Clients operate in an open-loop model: a client does not wait for the decision of a submitted value before submitting a new one. The rate at which clients submit values to Paxos is an experiment parameter, and all clients submit values at the same rate.

3.4.2 Performance

Figure 3.3 compares the performance of Paxos in the three setups: Baseline, Gossip, and Semantic Gossip. Experiments were carried out in AWS with distinct numbers of Paxos processes: $n = 13$, 53, and 105. These system sizes were obtained by placing, respectively, 1, 4, and 8 processes in each of the 13 AWS regions. An additional process, acting as the Paxos coordinator, was placed in the North Virginia region to achieve $n = 53$ and 105. In all experiments, the load is

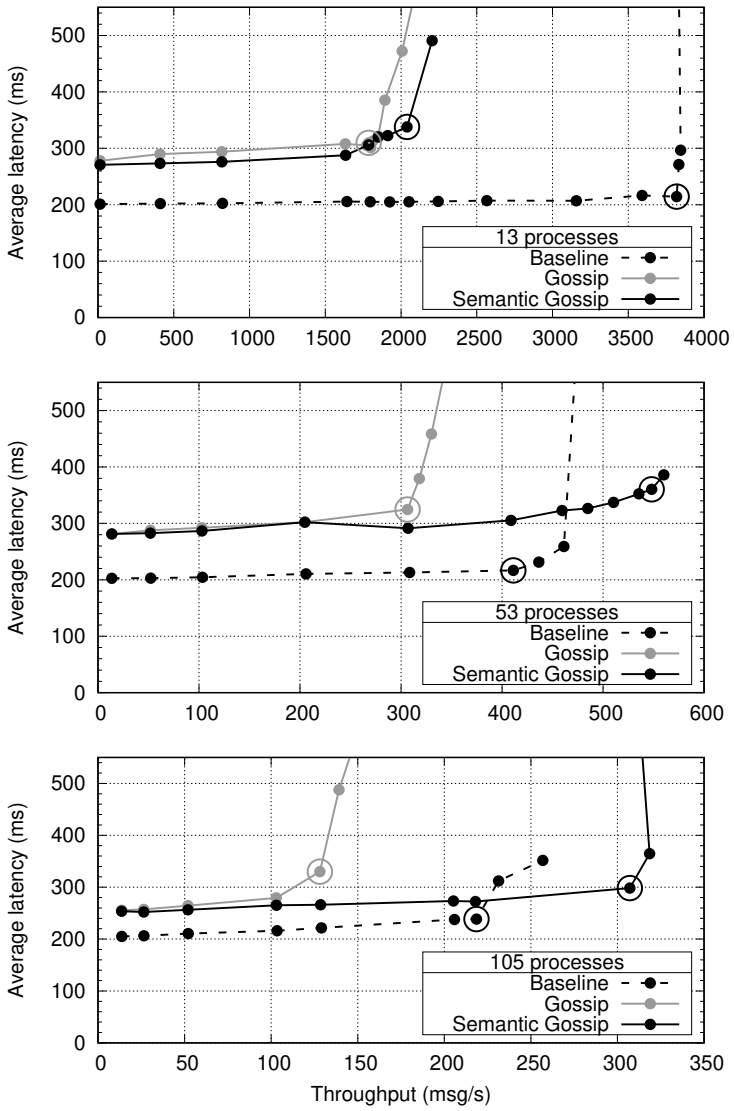

Figure 3.3. Overall performance of Baseline, Gossip and Semantic Gossip, with varying system sizes and 1KB values.

generated by 13 clients, one per region, that submit values at a fixed rate. We ran experiments with distinct value sizes, but we only present data for 1KB values, because results with other values sizes presented similar trends. We subjected Paxos to increasing client workloads (submission rates) until we noticed that the protocol is saturated. We highlight the saturation points in the graphs by drawing a circle around them. More precisely, for each setup and system size we highlight the point of the highest ratio between average latency and throughput. From this

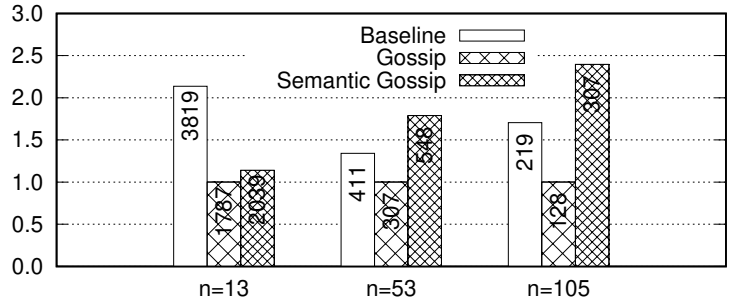

Figure 3.4. Normalized throughput at saturation point in the three setups. Absolute throughput (messages per second) presented in the bars.

point on, increasing client workloads results in small throughput increments at the cost of relevant latency increments. Saturation throughputs, normalized for the system size, are summarized in Figure 3.4.

The first conclusion we can draw from Figure 3.3 is the relevant overhead derived from the adoption of gossip as a communication means in a partially connected network. In fact, the multiple communication hops required to deliver messages to their destinations via gossip results in a relevant increment in the average latencies to order values, when compared to the Baseline setup, where we artificially assume full connectivity. When considering the lowest workload, the left-most points in the graphs of Figure 3.3, the average latencies in the Gossip setup are 38%, 39%, and 25% higher than in the Baseline setup for $n = 13$, 53, and 105. As we increase the workload, the overhead due to the adoption of gossip communication grows, so that in the saturation points of the Gossip setup average latencies are 51%, 52%, and 49% higher than in the Baseline setup, for $n = 13$, 53, and 105. In addition, we observe that Paxos in the Gossip setup saturates before, i.e., at lower workloads than in the Baseline setup. As a result, throughputs at the saturation points in the Gossip setup are, for $n = 13$, 53, and 105, respectively, 47%, 74%, and 59% lower than in the Baseline setup.

An explanation for the performance degradation of Paxos in the Gossip setup is the inherent redundancy of gossip communication. We then compared the number of messages received by the Paxos coordinator in Baseline and Gossip setups. In the Baseline setup the coordinator is the only process that communicates directly with all processes, thus the most overloaded process. In the Gossip setup, in terms of communication, the coordinator is a process like any other. With $n = 105$ processes, the number of messages received by a regular process in the Gossip setup is around 8 times the number of messages received by the coordinator in the Baseline setup. In fact, the gossip layer discards around 87%

(about 7/8) of received messages because they are duplicated. For smaller system sizes the redundancy factor observed in the Gossip setup is smaller but still relevant. For $n = 53$, the redundancy factor is about 5 times and around 80% of received messages are duplicated. For $n = 13$, the redundancy factor is about 2 times and around 49% of received messages are duplicated. This difference is due to the average number of processes to which each process is connected, of the order of $log_2 n$ ($log_2 105 \approx 6.7$, $log_2 53 \approx 5.7$, and $log_2 13 \approx 3.7$).

A second observation from Figure 3.3 is the performance improvement obtained with the adoption of the semantic filtering and aggregation techniques. For the smallest system size, $n = 13$, and workloads below the saturation of Paxos, we observe a discrete but consistent reduction in average latencies in the Semantic Gossip setup when compared with the Gossip setup: from 6% to 7%. Then, around the saturation workload of the Gossip setup, the behavior in the Gossip and Semantic Gossip setups become quite similar, although the saturation throughput in the Semantic Gossip setup is 14% higher than in the Gossip setup. With $n = 53$, despite some fluctuation in results, we note an overall performance improvement derived from the adoption of the semantic techniques. At the Gossip setup's saturation workload, in particular, the average latency is 11% lower in the Semantic Gossip setup, which also reaches a saturation throughput 79% higher than in the Gossip setup. The improvement is more noticeable for $n = 105$, where the corresponding reduction in average latency reaches 24% while the increase in the saturation throughput is of 2.4× with Semantic Gossip.

The advantage of Semantic Gossip can be explained when we compare the number of messages exchanged by processes via gossip. With $n = 105$, considering the saturation point of the Gossip, the number of messages received by a process in the Semantic Gossip setup is 58% lower than in the Gossip setup. To this reduction, contribute both the messages discarded through semantic filtering and multiple messages replaced by a single message through semantic aggregation. If we consider the messages delivered to Paxos (when received for the first time and possibly disaggregated), the number is 16% lower in the Semantic Gossip setup, as a direct result of semantic filtering. The portion of messages discarded because they are duplicated is 82% in the Semantic Gossip setup, a small reduction from the 87% observed in the Gossip setup. The inherent redundancy of gossip communication is thus preserved, just as Paxos still operates with a reasonably safe level of redundancy.

Latency distributions. Figure 3.5 presents the cumulative distribution function (CDF) of latencies measured by clients in a given configuration for the three

analyzed setups. The data presented refers to experiments with $n = 105$ and the same client workload (104 submissions/s), the biggest workload under which the protocol is not yet saturated in the three setups.

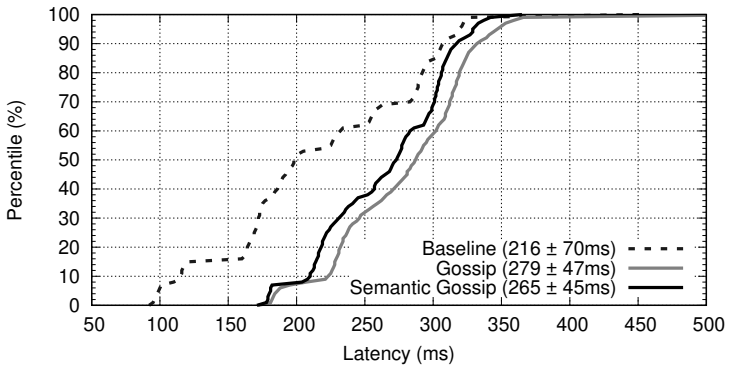

Figure 3.5. Latency distribution in all setups with $n = 105$. Legend with average latency and standard deviation.

As observed in Figure 3.5, latency distributions show considerable dispersion and present several noticeable steps. This occurs because latencies are measured by 13 clients, one client per region, that submit a value to a Paxos process located at the same region, then wait until the corresponding decision is informed by the same process. The client located at the same region as the coordinator has the advantage of having its values delivered to the coordinator with low delays. Latencies measured by this client, about 7.7% (1/13) of all, are the lowest, noticeable in the bottom left part of the CDFs. Values submitted by clients located in other regions are forwarded to the coordinator, an operation subjected to WAN latencies. The cost of this operation is more noticeable in curves for the Baseline setup, as Paxos processes are, exceptionally, allowed to send values directly to the coordinator. From the second region (Canada) to the coordinator's region (North Virginia) the latency is relatively small: 7ms. The second step in Baseline's CDF is thus around 15.4% (2/13). Then, up to the seventh region (Frankfurt), WAN latencies are larger but still below 50ms. The step around 53.8% (7/13) represents this interval, corresponding approximately to the median of the latency distributions in the Baseline setup.

Latencies observed by a client are less affected by its geographic location in the Gossip and Semantic Gossip setups than in Baseline setup. As a result, the standard deviation of latencies is lower in the Gossip and Semantic Gossip setups than in the Baseline setup. The least latency variability in gossip-based setups is associated to the adoption of a randomly generated overlay network. While pro-

cesses located in close geographical regions are not necessarily connected, which increases latency between them, processes farther from the coordinator are not so significantly penalized. In fact, from the 70th-percentile, corresponding to latencies measured by the 4 clients more distant from the coordinator, the overhead imposed by the adoption of gossip communication is much less noticeable (less than 20ms or 6%).

When comparing the Gossip and Semantic Gossip setups, we observe an almost constant distance between the CDFs. Except for the latencies measured by clients connected to the Paxos coordinator, from the 7th to the 97th percentiles latencies measured in the Semantic Gossip setup are from 13ms to 20ms (5.0% to 5.6%) lower than in the Gossip setup. The average latency in the Semantic Gossip setup is 5.4% lower than in the Gossip setup. The improvement in average latencies reaches 24% in the saturation point of the Gossip setup, but we choose to compare the setups under a workload at which none of them is saturated. A less noticeable aspect in Figure 3.5 is the tail of the latency distributions. The 99.9th latency percentile in the Semantic Gossip setup is 140ms (28%) lower than in the Gossip setup; which, in its turn, is 54ms lower than in the Baseline setup. In addition to the lowest latency standard deviation, this reaffirms the less variable latencies observed in the Semantic Gossip setup.

3.4.3 Reliability

A major feature of gossip-based communication is its reliability, which allows masking link and process failures. This capability stems from the inherent redundancy of gossip communication, attested by the data collected in our previous experiments. Since a message is transported through multiple distinct paths in the overlay network, a communication disruption between two processes is less likely to prevent the message from being received by all destinations. In this section, we assess the degree of reliability provided by the Gossip and Semantic Gossip setups.

We implemented a fault-injection mechanism that randomly discards messages received by a process. In addition, the timeout-trigged procedures that enable Paxos to react to message loss events were disabled. As a result, Paxos processes may fail to learn the decision for some consensus instances. The impact for the clients is more relevant: a single unsuccessful instance of consensus renders all subsequent instances in the same execution also unsuccessful, as values are delivered in total order, with no gaps. As clients operate in an open-loop, they continue submitting values at a given rate even after failing to order a value. We can then compute the number of values that were submitted by clients but

not ordered by Paxos, due to the injected message loss.

Figure 3.6 summarizes the impact of message loss in the operation of Paxos with $n = 105$ processes in the Gossip and Semantic Gossip setups. We subjected Paxos to increasing workloads, the number of values submitted per second by the 13 clients (y axis), and increasing injected message loss rates (x axis). We ran 10 experiments for each client workload and message loss rate to minimize the effect of particularly favorable or unfavorable executions (as messages are discarded at random). Due to the large number of executions, the experiments were carried out in the emulated AWS environment. Figure 3.6 depicts the aggregate portion of values submitted but *not ordered* in each configuration. The white cells of the graph represent configurations in which all submitted values were successfully ordered in the 10 executions, despite the injected message loss (i.e., we omit the 0% values).

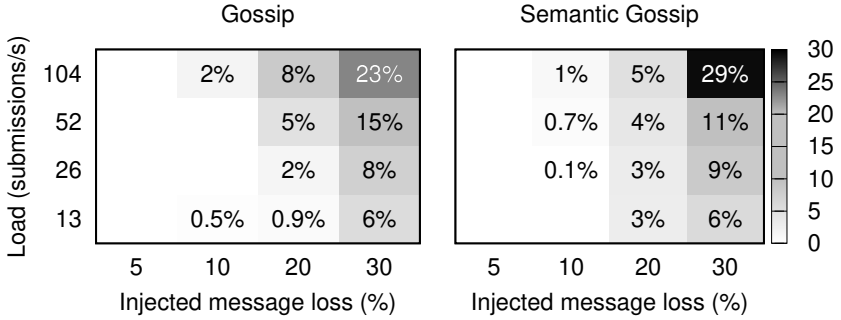

Figure 3.6. Impact of message loss in the reliability of Paxos in the Gossip and Semantic Gossip setups under injected message loss, as the portion of failed instances of consensus.

We can draw two major conclusions from Figure 3.6. First, the Gossip setup is indeed resilient to message loss: with injected message loss rates below 10% every submitted value is ordered. This means that: (i) every submitted value was received by the coordinator; (ii) at least a majority of processes received the Phase 2a message from the coordinator and accepted the submitted value; and (iii) all 13 processes handling clients received both the Phase 2a message and Phase 2b messages from a majority of processes. With 10% of injected message loss, only 2% of the submitted values were not ordered, due to the violation of any of the three conditions above mentioned. Notice that potentially less than 2% of the consensus instances have actually failed, but the client did not deliver any values ordered after the first unsuccessful instance of consensus. With 20% of message loss, up to 8% of instances of consensus are affected, while when

30% are discarded up to 23% of the submitted values are not ordered—in both cases under more than 100 submissions/s workload. Second, the benefits on performance obtained with the adoption of the semantic extensions do not come at the cost of lower reliability. In fact, the Semantic Gossip setup has proved to be, in the overall, as reliable as the Gossip setup under message loss rates up to 20%. With 30% of message loss and under higher workloads, however, the portion of values that Paxos failed to order reaches 29% in the Semantic Gossip setup. This indicates that under such (extreme) circumstances the semantic extensions may impact the inherent reliability of gossip communication.

3.4.4 Network overlays

The overlay network interconnecting the processes, and in particular the latencies between the coordinator and the remaining processes, affects the performance of Paxos. In fact, since the decision of a value requires a round-trip from the coordinator to a majority of processes, the median of RTTs from the coordinator to other processes ultimately dictates the latency of a Paxos instance. Distinct random overlay networks are likely to present different medians of RTTs from the coordinator to other processes, and so will have different baseline latencies for deciding values. This is the reason for enforcing the same network overlay in all experiments in the Gossip and Semantic Gossip setups with the same system size, as mentioned in Section 3.4.1.

Figure 3.7 illustrates the method to select the overlay network enforced in experiments with the same system size, $n = 105$ in the case. We randomly generated 100 network overlays and submitted them to a minimal client workload in the Gossip setup. For each network overlay, we compute the median of RTTs from the coordinator to all processes (x axis), and associate it with the obtained latency (y axis). Notice that multiple overlay networks can have the same median RTT but sport distinct latencies, as the RTT is not the only element to determine latencies. These two parameters allow totally ordering the multiple overlay networks, from which we select the median one. Due to the high number of executions required, we carried out these experiments in the emulated AWS environment. Once the overlay network is selected, we enforce it in AWS and verify whether the performance in the real environment is similar. Figure 3.7 highlights the selected overlay network, and presents the latency achieved in this overlay network in both emulated and actual AWS EC2 environments.

The adoption of a single network overlay for all core experiments with Gossip and Semantic Gossip setups raises another research question: Are the performance improvements observed with the adoption of the semantic techniques

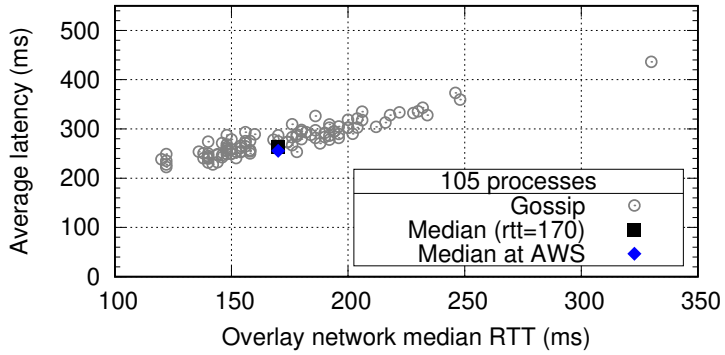

Figure 3.7. Latency of Paxos in the Gossip setup under low workload in 100 distinct overlay networks. The overlay network adopted in the core experiments is highlighted.

associated with the choice of a particular overlay network? To answer this question, we selected a client workload at which the Gossip setup becomes saturated, and adopted this workload to assess the Paxos performance in the Gossip and Semantic Gossip setups in 100 distinct overlay networks. Due to the high number of executions involved, experiments were carried out in the emulated AWS environment.

Figure 3.8 presents results with $n = 105$ processes in the Gossip and Semantic Gossip setups, adopting the same 100 overlay networks illustrated in Figure 3.7. We aggregate data of overlay networks with the same median RTT (x axis), presenting the average latency (y axis) among multiple experiments, for the sake of readability; Figure 3.8 therefore present 44 data points for each setup. The workload applied to Paxos in these experiments is enough to evidence the performance improvements derived from the adoption of the semantic techniques. In fact, for all network overlays considered in Figure 3.8, Semantic Gossip improves latency from 11% to 39%, 23% on average, when compared to the Gossip setup. As a reference, the improvement observed in the network overlay adopted in the core experiment is of 24% from the Semantic Gossip to the Gossip setups.

3.4.5 Summary

In this section, we recall the main conclusions from our study.

- The adoption of gossip communication has a negative impact on the performance of Paxos. While this is an expected result, our study quantifies

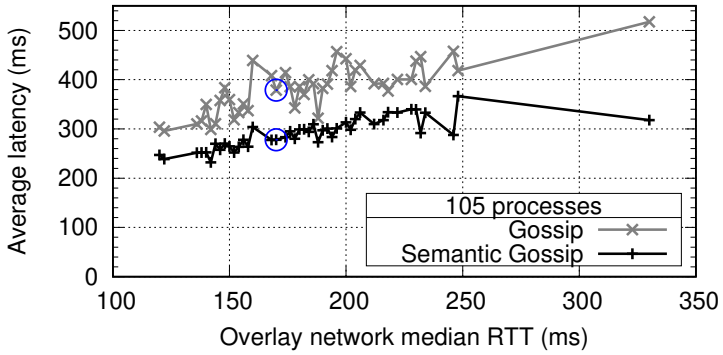

Figure 3.8. Latency of Paxos in the Gossip and Semantic Gossip setups in 100 distinct overlay networks. The overlay network adopted in the core experiments is highlighted.

this overhead: Gossip slows down the decision of values, by increasing by up to 52% the average latencies when compared with the Baseline setup.

- Augmenting the gossip-based communication substrate with semantic extensions improves the performance of Paxos in all configurations evaluated: Semantic filtering and aggregation reduce the number of messages exchanged by processes via gossip by up to 58%. As a result, Semantic Gossip provides average latencies from 7% to 24% lower than in the Gossip setup, while sustaining up to 2.4× higher workloads and providing stable and less variable latencies.

- The proposed semantic extensions do not compromise the reliability of gossip communication. For example, without any timeout-trigged retransmission mechanisms, Paxos was able to operate correctly despite up to 10% of message loss, both in the Gossip and Semantic Gossip setups.

3.5 Conclusion

This chapter investigates the deployment of consensus protocols in partially connected networks that rely on gossip communication. We introduce Semantic Gossip, a gossip-based communication substrate that takes consensus semantics into account to optimize performance.

Semantic Gossip relies on two techniques, semantic filtering and semantic aggregation. With semantic filtering, the gossip protocol can stop propagating messages that have become redundant from the perspective of the consensus

protocol. With semantic aggregation, the gossip protocol can replace multiple consensus protocol messages by a single message of equivalent meaning.

Both techniques reduce the number of messages that are propagated by gossip without penalizing the resilience of gossip communication. We have demonstrated the usefulness of Semantic Gossip using Paxos, a well-known consensus protocol.

We believe that other agreement protocols could also benefit from a gossip-based communication substrate with semantic extensions. First, semantic filtering is motivated by the fact that some messages in Paxos supersede previous messages. Whenever this happens, superseded messages can be dropped without negative consequences for the protocol. This is an aspect that is not particular to Paxos, but present in other agreement protocols (e.g., based on rounds). Second, semantic aggregation is inspired by a common pattern in agreement protocols where a protocol step depends on votes cast by processes in previous steps. Instead of sending all votes to all processes, votes can be aggregated and propagated as one message. Finally, since gossip communication offers probabilistic delivery guarantees, agreement protocols that can cope with message loss would be more appropriate to the proposed techniques.

Chapter 4

How robust are synchronous consensus protocols?

4.1 Introduction

The synchronous system model requires every message to be delivered within a predefined time-bound Δ. To increase the probability of this happening, existing synchronous consensus protocols set Δ as the 99.99-th percentile of sampled communication [80] or as a 10-time factor of average latency [3]. The reliance on Δ influences protocol correctness and directly impacts its performance. The result is a challenging tradeoff: a conservative Δ reduces the chances of synchrony violations, which favors correctness, but results in poor protocol performance.

In this chapter, we delve into this tradeoff. Our starting point is the observation that some synchronous consensus protocols can tolerate synchrony violations without compromising correctness. Resilience to synchrony violations happens due to communication diversity and redundancy in a protocol. In Figure 4.1 (left), process p sends a request to process q (m_A) and sets a 2Δ timeout for the answer from q. Even if p's request violates synchrony (i.e., m_A takes longer than Δ to arrive at q), q's response (m_B) makes up for the delay and arrives at p within the expected 2Δ. In Figure 4.1 (right), p sends a request to q and r (m_A) and sets a 3Δ timeout for their answer. Process q receives m_A timely, replies to p (m_B) and relays m_A to r. Although r receives m_A from p after Δ, it receives m_A from q timely and responds to p (m_C). As a result, p receives responses from q and r within the expected 3Δ. These communication patterns are at the core of BoundBFT, a novel Byzantine fault-tolerant (BFT) synchronous consensus protocol introduced in this chapter.

Tolerating even a few synchrony violations can provide substantial gains in

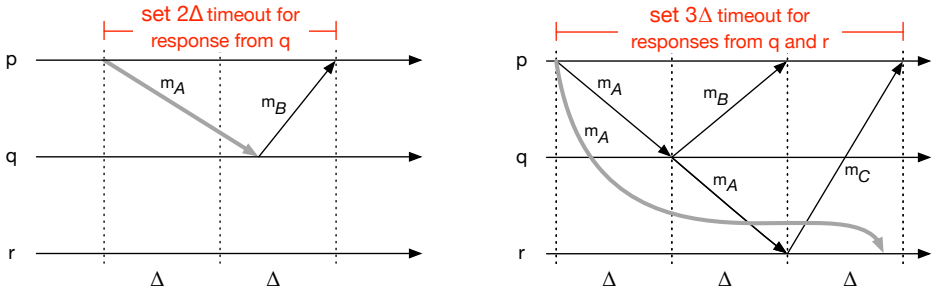

Figure 4.1. Possible BoundBFT execution patterns where several messages (in gray) violate synchronous bound Δ without compromising protocol correctness. Some messages are omitted to avoid cluttering.

	US West (CA)		Europe (EU)		Tokyo (JP)		Sydney (AU)		Sao Paolo (BR)	
	99.99%	99.999%	99.99%	99.999%	99.99%	99.999%	99.99%	99.999%	99.99%	99.999%
US East (VA)	1097	82190	1112	85649	1226	81177	1372	95074	1214	85434
US West (CA)			1184	1974	1133	1180	1209	6354	1252	90980
Europe (EU)					1310	1397	1375	3154	1257	1382
Tokyo (JP)							1149	1414	2496	11399
Sydney (AU)									1496	2134

Table 4.1. Round-trip latency (in milliseconds) of hping3 across Amazon EC2 datacenters, collected during three months [80].

performance. To understand why, consider Table 4.1, reproduced from [80], which compares the 99.99-th and 99.999-th percentile of communication across Amazon EC2 datacenters. For example, a synchronous protocol that can tolerate one synchrony violation in every ten thousand messages exchanged between US West and US East can benefit from a significantly lower timeout than a protocol that expects one synchrony violation in every one hundred thousand messages to be transmitted timely (i.e., 1097 versus 82190 milliseconds).

Since BoundBFT tolerates Byzantine failures, synchrony violations should not introduce vulnerabilities that malicious processes could exploit. In leader-based consensus protocols that tolerate Byzantine failures, such as BoundBFT, the leader is the most advantageous role for a malicious process as it can induce honest processes into inconsistent decisions, possibly with help from other malicious processes. In a synchronous protocol, the malicious leader can hope to get "additional help" from synchrony violations, as some honest processes may be delayed with respect to other processes. The attack will work as long as the deceived honest process does not find out about the trickery before deciding. However, honest processes communicate with many other honest processes, so there is ample opportunity to find out about the attack even if some messages are delayed.

In this chapter, we investigate the robustness of BoundBFT, by which we mean its ability to maintain correctness under synchrony violations, both in the presence and absence of malicious processes. Building on BoundBFT's leader-based execution model with signed messages, we first characterize the range of potential attacks and examine their effects when combined with synchrony violations. Based on this characterization, we identify specific attacks and implement them to rigorously evaluate BoundBFT's robustness. Using these implemented attacks, we conduct experiments to determine an appropriate synchrony bound, Δ, that provides high confidence in preserving protocol correctness, even when malicious replicas are present. Finally, we apply this bound to evaluate BoundBFT's performance, allowing for a meaningful comparison with partially synchronous protocols.

We have implemented BoundBFT and compared it to state-of-the-art synchronous (i.e., Sync HotStuff [3; 5]) and partially synchronous consensus protocols (i.e., Tendermint [20] and HotStuff-2 [82]). Our evaluation in an emulated geographically distributed system showed that BoundBFT's synchrony bounds could be, in some cases, more than one order of magnitude smaller than typical synchronous consensus protocols without correctness violations under attack. Consequently, BoundBFT sports latency comparable to partially synchronous consensus protocols, much smaller than the latency of synchronous counterparts. Finally, BoundBFT outperforms considered partially synchronous consensus protocols with increased reliability and availability.

4.1.1 Outline

The remainder of the chapter is structured as follows. Section 4.2 presents BoundBFT, a new BFT consensus algorithm designed for the synchronous system model. Section 4.3 analyzes BoundBFT under synchrony violations and attacks. Section 4.4 experimentally evaluates BoundBFT and competing approaches, and Section 4.5 concludes the chapter.

4.2 BoundBFT

This section introduces BoundBFT, a new consensus algorithm introduced in this thesis. This chapter is divided into three parts: the detailed algorithm description, an intuitive explanation of the algorithm's correctness, and a formal proof of its correctness.

4.2.1 Protocol overview

BoundBFT is a synchronous BFT consensus protocol with a rotating leader [5]. It draws inspiration from Tendermint [20], a partially synchronous consensus protocol, and the rotating-leader version of Sync Hotstuff [5; 3], a synchronous consensus protocol.

BoundBFT adopts the good-case execution of Sync HotStuff, achieving optimal latency and responsive leader rotations [5], and Tendermint's termination mechanism, which does not require an explicit leader change subprotocol. BoundBFT improves on Sync HotStuff by reducing the waiting time for a new leader to propose from 9Δ to 5Δ when the previous leader is silent. Furthermore, unlike the original Tendermint protocol, BoundBFT does not rely on a gossip communication layer to reliably disseminate all messages to replicas. Lastly, BoundBFT can tolerate $f < n/2$ Byzantine replicas, compared to Tendermint's tolerance of $f < n/3$.

Variables

Algorithm 1 presents a set of variables that each honest replica maintains. BoundBFT's execution evolves as a sequence of epochs, numbered $0, 1, 2, \ldots$, with each replica tracking the last epoch it started, denoted as e_p. Each epoch e has a designated leader, computed using a deterministic function $leader(e)$. The leader is responsible for proposing a new block in the epoch. Replicas vote once per epoch for the first leader proposal they receive. The flag $hasVoted_p$ indicates whether the replica has already voted in the current epoch. When a replica receives $f+1$ votes signed by distinct replicas for a block b proposed by the leader in epoch e, it forms a block certificate $C_e(b)$ with the $f+1$ signed votes. Each replica maintains the most recent certified block and its certificate in variables $validBlock_p$ and $validBC_p$, respectively. A certificate $C_e(b)$ is considered more recent than $C_{e'}(b')$ if $e > e'$. Additionally, if a replica receives a block certificate for a block proposed in the current epoch before detecting any misbehavior, it locks on this block by setting the $lockedBlock_p$ and $lockedBC_p$ variables. Finally, each replica tracks whether the epoch is still active or finished with the variable $epochState_p$. An epoch finishes when a replica commits a block or receives proof of leader misbehavior.

Normal execution

Algorithm 2 presents BoundBFT's pseudo-code that covers executions when leaders are honest. At the start of an epoch, the leader l broadcasts the proposal

Algorithm 1 BoundBFT consensus algorithm: variables

```
1: Initialization:
2:   e_p := 0                                   ▷ the current epoch
3:   hasVoted_p := false                        ▷ has the replica voted in the current epoch?
4:   validBC_p := nil                           ▷ the most recent block certificate the replica is aware of and...
5:   validBlock_p := nil                        ▷ the block certified by validBC_p
6:   lockedBC_p := nil                          ▷ the block certificate the replica is locked on and...
7:   lockedBlock_p := nil                       ▷ the block certified by lockedBC_p
8:   epochsState_p[] := nil                     ▷ an epoch can be in one of the states: ACTIVE, COMMITTED, NOT-COMMITTED
                                                ▷ ACTIVE - replica is in this state until it commits a block or detects a misbehavior
                                                ▷ COMMITTED - replica committed a block before detecting any misbehavior
                                                ▷ NOT-COMMITTED - replica detected a misbehavior before committing a block
```

Algorithm 2 BoundBFT consensus algorithm: normal case

```
1: when bootstrapping do StartEpoch(0)          ▷ the execution starts in epoch 0
2: Procedure StartEpoch(e) :                    ▷ upon starting an epoch:
3:   e_p ← e                                    ▷ the replica resets the current epoch variables
4:   epochsState_p[e_p] ← ACTIVE
5:   hasVoted_p ← false
6:   if leader(e_p) = p then                    ▷ if the replica is the leader in the current epoch...
7:     block.txs ← GetTxs()                     ▷ it gets new transactions to include in the new block
8:     if validBlock_p ≠ nil then               ▷ then, if it knows of a previously certified block...
9:       block.prev ← id(validBlock_p)          ▷ it links the new block with that block
10:      broadcast ⟨PROPOSE, e_p, block, validBC_p⟩_p   ▷ lastly, it broadcasts the proposal with the new block and....
                                                ▷ the certificate for the block it is extending, validBC_p
11: when receive ⟨PROPOSE, e, b, BC⟩_l where valid(b) and   ▷ upon receiving the valid proposal...
12:  l = leader(e) and e = e_p do               ▷ from the leader of the current epoch:
13:    if epochsState_p[e] = ACTIVE ∧ b.prev = BC.id ∧ hasVoted_p = false ∧   ▷ if the epoch is active, replica has not voted yet, and...
14:    BC.epoch ≥ lockedBC_p.epoch then         ▷ the proposed block extends a block at least as recent as replica's lockedBC...
15:      broadcast ⟨VOTE, e_p, id(b)⟩_p         ▷ the replica votes for a proposal, VOTE message contains block's hash
16:      hasVoted_p = true                      ▷ then, the replica sets hasVoted_p so it does not vote twice, and...
17:      forward ⟨PROPOSE, e, b, BC⟩_l          ▷ forwards the proposal message
18: when receive ⟨PROPOSE, e, b, BC⟩_l and f + 1 distinct ⟨VOTE, e, id(b)⟩_*   ▷ when the replica receives a proposal and f + 1 votes...
19:  where e = e_p do                           ▷ from the current epoch:
20:    cert ← NewCert from f + 1 ⟨VOTE, e, id(b)⟩_*   ▷ it forms a block certificate
21:    if epochsState[e] = ACTIVE then          ▷ if no misbehavior is noticed in the current epoch...
22:      lockedBC_p ← cert                      ▷ the replica locks on this block by setting lockedBC_p to cert and...
23:      lockedBlock_p ← b                      ▷ lockedBlock_p to b, and...
24:      start timeoutCommit(e_p, b)            ▷ starts timeoutCommit
25:    validBC_p ← cert                         ▷ the replica always updates its validBC_p and validBlock_p...
26:    validBlock_p ← b                         ▷ to the most recent block
27:    forward messages from cert               ▷ lastly, the replica forwards the votes to other replicas and...
28:    StartEpoch(e + 1)                        ▷ starts the next epoch
29: when timeoutCommit(e, b) expires do         ▷ when timeoutCommit expires and...
30:    if epochsState[e] = ACTIVE then          ▷ the replica did not observe any proof of misbehavior,
31:      epochsState[e] ← COMMITTED             ▷ the replica commits the block b and...
32:      CommitBlockAndItsAncestors(b)          ▷ all its already uncommited ancestor blocks
```

containing a new block b that extends the most recently certified block it knows of, $validBlock_l$ (lines 6–10 in Algorithm 2). Along with the new block, the leader includes the certificate for $validBlock_l$, $validBC_l$.

Upon receiving a proposal (lines 11–17 in Algorithm 2), a replica verifies the proposal's validity (see Section 2.3) and votes for it if the proposed block truely extends the block from the leader's block certificate, if the leader's block certificate is at least as recent as the replica's $lockedBC_p$. The replica votes by

sending a signed vote message to all replicas. A vote contains the current epoch number and the hash of the block, $id(b)$.

When a replica receives a proposal and $f+1$ votes for it, it forms a block certificate for the proposed block. If the replica has no proof of leader l misbehaving, it locks on b and triggers $timeoutCommit(e, b)$ (lines 18–24 in Algorithm 2). The replica then updates its $validBlock_p$ and $validBC_p$ variables (lines 25–28 in Algorithm 2) and starts epoch $e+1$. To ensure all honest replicas receive the proposal and its certificate, the replica forwards them (lines 17 and 27 in Algorithm 2), allowing all honest replicas to start epoch $e+1$ within Δ time.

When $timeoutCommit(e, b)$ expires and the replica has no evidence of leader misbehavior for epoch e, it commits block b and all blocks b extends (lines 29–32 in Algorithm 2). In other words, it directly commits block b and indirectly commits all its uncommitted ancestor blocks.

The replica does not wait for $timeoutCommit(e, b)$ to expire before starting the next epoch. Instead, it begins epoch $e+1$ immediately after receiving a block certificate in epoch e (line 28 in Algorithm 2). This approach allows BoundBFT to change leaders without waiting for the conservative network delay Δ when we have a sequence of honest leaders, a property known as *optimistic responsiveness* [82]. Additionally, BoundBFT implements *pipelining* [119], enabling replicas to start working on the next block before committing the previous one. Specifically, the leader in epoch $e+1$ will propose a new block once it receives a block certificate for a block in epoch e.

Handling malicious leaders

Algorithm 3 presents BoundBFT's pseudo-code responsible for handling Byzantine leaders.

To detect a malicious leader, a replica r starts a timer, $timeoutCertificate(e)$, when it enters epoch e (line 2 in Algorithm 3). If $timeoutCertificate(e)$ expires and r is still in epoch e, it indicates that r did not receive a block certificate, which can only occur if the leader is Byzantine. Consequently, replica r blames the leader and broadcasts a message $\langle \text{BLAME}, e \rangle_r$ (lines 3–5 in Algorithm 3).

When a replica receives $f+1$ blame messages for epoch e from distinct replicas, it has proof that at least one honest replica blamed the leader and forms a blame certificate $C_e(\text{BLAME})$ (lines 6–7 in Algorithm 3).

Additionally, if an honest replica receives proposals for two distinct blocks signed by the leader in the same epoch e, it has proof that the leader is misbehaving. The replica then constructs an equivocation certificate $C_e(\text{EQUIV})$ (lines 9–11 in Algorithm 3).

Algorithm 3 BoundBFT consensus algorithm: handling malicious leaders

```
1:  upon starting the epoch e do                                   ▷ when a replica enters a new epoch:
2:      start timeoutCertificate(e_p)                              ▷ it starts the timer used to detect a malicious leader
3:  when timeoutCertificate(e) expires do                          ▷ when timeoutCertificate expires:
4:      if e = e_p ∧ epochsState[e] = ACTIVE then                  ▷ if the replica did not receive a block certificate and the epoch is still ACTIVE
5:          broadcast ⟨BLAME, e_p⟩_p                               ▷ the replica blames the leader by broadcasting a BLAME message
6:  when receive f + 1 distinct ⟨BLAME, e⟩_* do                     ▷ when receiving f + 1 distinct BLAME messages from an epoch:
7:      cert ← NewCert from f + 1⟨BLAME, e⟩_*                       ▷ the replica forms a blame certificate and...
8:      MissbehaviorDetected(cert, e)                              ▷ calls MissbehaviorDetected with the certificate and epoch as parameters
9:  when receive ⟨PROPOSE, e, b, BC⟩_p and ⟨PROPOSE, e, b′, BC′⟩_p  ▷ when replica receives two proposals...
10:     where p = leader(e) and b ≠ b′ do                          ▷ from leader for two distinct blocks:
11:     cert ← NewCert from ⟨PROPOSE, e, b, BC⟩_p and ⟨PROPOSE, e, b′, BC′⟩_p   ▷ the replica forms an equivocation certificate and...
12:     MissbehaviorDetected(cert, e)                              ▷ calls MissbehaviorDetected with the certificate and epoch as parameters
13: Procedure MissbehaviorDetected(cert, e) :                      ▷ when misbehavior is detected in an epoch:
14:     if epochsState[e] = ACTIVE then                            ▷ if the epoch is still active
15:         epochsState[e] ← NOT-COMMITTED                         ▷ the replica sets state to NOT-COMMITTED
16:         if e = e_p then                                        ▷ if cert is the first certificate for the current epoch,
17:             forward messages from cert                         ▷ the replica forwards the messages from certificate and...
18:             start timeoutEpochChange(e_p)                      ▷ triggers timeoutEpochChange
19:  when timeoutEpochChange(e) expires do                         ▷ when timeoutEpochChange expires:
20:     if e = e_p then                                            ▷ if the replica is in epoch e
21:         StartEpoch(e_p + 1)                                    ▷ the replica starts the next epoch
```

Whenever a replica has proof of the leader's misbehavior (i.e., a blame or equivocation certificate), it calls the function $MissbehaviorDetected(cert, e)$ and forwards the certificate and epoch number to it (lines 8 and 12 in Algorithm 3). If a block is not committed in epoch e, the replica marks the epoch state as NOT-COMMITTED (line 15 in Algorithm 3). Moreover, if $cert$ is the first certificate in epoch e, the replica forwards the certificate and triggers $timeoutEpochChange(e)$ (lines 16–18 in Algorithm 3). Forwarding the certificate ensures that all honest replicas learn that the leader is Byzantine within Δ time. Additionally, the extra timeout allows the replica to learn if an honest replica r moved to the next epoch before detecting leader misbehavior, i.e., r received a block certificate in epoch e, locked on it, and moved to the next epoch.

When $timeoutEpochChange(e)$ expires and the replica is still in epoch e, it moves to epoch e+1 (lines 19–21 in Algorithm 3). Replicas wait for $timeoutEpochChange$ before moving to the next epoch only in the case of a Byzantine leader. If the leader is honest, replicas form a block certificate and move to the next epoch without waiting for any timeouts.

4.2.2 Correctness intuition

In this section, we provide an intuitive explanation of BoundBFT's correctness. We structure the discussion around the three main properties guaranteed by BoundBFT.

Epoch synchronization

The epoch synchronization mechanism guarantees that honest replicas progress through each epoch in a coordinated manner. Specifically, all honest replicas initiate each epoch within Δ time, ensuring synchronization. Additionally, Byzantine replicas cannot disrupt or halt the protocol during any epoch.

The epoch synchronization mechanism in BoundBFT relies on certificates: to start a new epoch, a certificate (i.e., block, blame, or equivocation certificate) must be formed in the previous epoch. BoundBFT ensures that, regardless of Byzantine behavior, a certificate is created in each epoch.

The mechanism BoundBFT employs to guarantee the existence of a certificate is as follows: honest replicas initiate a $timeoutCertificate$ upon entering a new epoch (line 2 in Algorithm 2). If the timeout expires without receiving a certificate, the replica blames the leader. This results in two possible outcomes: (i) an honest replica forms one of the certificates, or (ii) no honest replica receives a certificate before the $timeoutCertificate$ expires. In case (ii), all $f+1$ honest replicas will blame the leader, resulting in the formation of a blame certificate.

Once a certificate is ensured for an epoch, synchronizing replicas becomes straightforward: each replica forwards the received certificate (lines 17 and 27 in Algorithm 2 and line 17 in Algorithm 3). Within Δ time, all honest replicas receive the certificate and start the next epoch if they have not already done so.

Agreement

BoundBFT ensures that no two honest replicas commit different blocks in the same blockchain height. Consequently, the resulting blockchain remains consistent and does not have forks.

In epochs with a Byzantine leader, multiple certificates can be created. As a result, different honest replicas may start the next epoch receiving different certificates. For instance, one honest replica may receive a block certificate, while another may receive an equivocation certificate. To account for this scenario, an honest replica commits a proposed block b in epoch e only if it knows that the first certificate received by all honest replicas in e is a certificate for b. This guarantees two properties: (i) all honest replicas vote for block b, and (ii) all honest replicas lock on block b in epoch e. Property (i) ensures that no other block can be certified and afterward committed in epoch e. Property (ii) guarantees that honest replicas vote only for blocks extending b in the following epochs. As a result, only b and blocks extending b will be certified and committed in epochs $e' \geq e$, and the agreement property will be satisfied.

The mechanism BoundBFT uses to verify the commit condition is as follows. Upon receiving a certificate for block b, $C_e(b)$, as the first certificate in epoch e, r forwards the certificate and triggers $timeoutCommit(e)$ at time t (lines 24 and 27 in Algorithm 2). Consequently, r knows that all honest replicas will receive $C_e(b)$ by time $t + \Delta$. If an honest replica p received a different certificate before $C_e(b)$, it must have received it at time $t_1 < t + \Delta$. Since p also forwards its certificate, r will receive it by time $t_1 + \Delta < t + 2\Delta$. Therefore, setting $timeoutCommit(e)$ to 2Δ ensures that r receives p's certificate on time. Ultimately, if $timeoutCommit(e)$ expires and r has not heard about any other certificates, r can be sure that the first certificate received by all honest replicas in e is $C_e(b)$. In this case, r commits block b.

Progress

BoundBFT ensures that all honest replicas commit a new block in every epoch with an honest leader. It does so by: (i) ensuring that all honest replicas vote for the leader's proposal, and (ii) preventing the creation of blame or equivocation certificates. Property (i) guarantees the creation of a unique block certificate, while property (ii) guarantees that all honest replicas must receive the block certificate, trigger $timeoutCommit$, and, when it expires, commit the proposed block.

An honest leader of an epoch proposes a new block that extends its $validBlock$ and sends $validBC$ together with the new block. Other honest replicas will vote for the new proposal only if the $validBC$ sent by the leader is at least as recent as their $lockedBC$. Consequently, BoundBFT ensures that whenever an honest replica locks on a block in epoch e, all honest replicas update the $validBlock$ and $validBC$ to the block certified in epoch e. Therefore, the $validBC$s on all honest replicas are always at least as recent as $lockedBC$s on all honest replicas.

BoundBFT ensures that $validBlock$ and $validBC$ are always up to date by relying on a mechanism that uses $timeoutEpochChange$. Namely, an honest replica r cannot start the next epoch immediately if the first certificate it receives in the current epoch is a blame or equivocation certificate. Instead, it must ensure no other honest replica locks on a block in this epoch. Consequently, r forwards its certificate (line 17 in Algorithm 3), knowing that in Δ time, all honest replicas will receive it. If any honest replica p locked on a block, it must have done so before receiving the forwarded certificate. As a result, upon forwarding its certificate, r sets $timeoutEpochChange(e)$ to expire in 2Δ time (line 18 in Algorithm 3). Moreover, r starts the next epoch only when this timeout expires, or it receives the block certificate for the current epoch. Since p also forwards the

certificate after locking (line 27 in Algorithm 2), r knows it will receive it before $timeoutEpochChange$ expires. Notably, r will not lock on a block certificate if it receives the certificate after $timeoutEpochChange$ is initiated.

Lastly, BoundBFT must ensure that no equivocation or blame certificates are possible in epochs with honest leaders. An equivocation certificate will not be formed since the honest leader will not propose two different blocks. However, ensuring that no blame certificate is possible requires that no honest replica blames the leader. In other words, every honest replica must receive the block certificate before $timeoutCertificate(e)$ expires. Consequently, honest replicas set $timeoutCertificate(e)$ to 3Δ. The first Δ accounts for epoch drift time, the second Δ for the time it takes for the leader's proposal to reach all honest replicas, and the last Δ is for the reception of the votes broadcast by honest replicas. Since we already showed that all honest replicas will vote for the honest leader, the block certificate is formed on all honest replicas before the $timeoutCertificate(e)$ expires, and no honest replica blames the leader.

4.2.3 Correctness proof

This section presents the proof that BoundBFT satisfies all the properties of blockchain protocol (Section 2.3).

Lemma 1. *Every honest replica always moves to the next epoch.*

Proof. Assume for contradiction that an honest replica r remains in some epoch e indefinitely. This would imply that r did not generate any of the certificates $C_e(B_k)$, $C_e(\text{BLAME})$, or $C_e(\text{EQUIV})$. However, each honest replica starts the timer, $timeoutCertificate(e)$, upon entering epoch e (line 2 in Algorithm 3). When this timeout expires, if an honest replica has not received any certificate, it broadcasts the BLAME message (lines 3–5 in Algorithm 3). Consequently, if no certificate is formed before the $timeoutCertificate(e)$ expires, all honest replicas will broadcast the BLAME message, leading to the formation of the blame certificate $C_e(\text{BLAME})$. This contradicts our assumption, thus proving that every honest replica moves to the next epoch. □

Lemma 2. *If an honest replica starts epoch e at time t, then all honest replicas start epoch e by time* $t + \Delta$.

Proof. Suppose an honest replica r starts epoch e at time t. This implies that r receives and broadcasts $C_{e-1}(B_k)$ at time t (lines 27–28 in Algorithm 2), or at time $t - \text{timeoutEpochChange}(2\Delta)$, r receives and broadcasts $C_{e-1}(\text{BLAME})$

or $C_{e-1}(\text{EQUIV})$ (lines 17 and 21 in Algorithm 3). Messages with certificates will arrive within Δ time. Consequently, in the former case, all honest replicas receive $C_{e-1}(B_k)$ by time $t + \Delta$ and start epoch e. In the latter case, all honest replicas receive $C_{e-1}(\text{BLAME})$ or $C_{e-1}(\text{EQUIV})$ by time $t-\Delta$ and within 2Δ they start epoch e, ensuring that all honest replicas start epoch e by time $t + \Delta$. □

Theorem 1. *(Epoch synchronization) All honest replicas continuously move through epochs, with each replica starting a new epoch within Δ time of any other honest replica.*

Proof. We prove this theorem by combining Lemma 1 and Lemma 2.

First, from Lemma 1, we know that every honest replica always moves to the next epoch. This ensures that no honest replica remains stuck in any epoch indefinitely.

Second, from Lemma 2, we know that if an honest replica starts epoch e at time t, then all honest replicas start epoch e by time $t + \Delta$. This guarantees that all honest replicas start each epoch within Δ time of each other.

Combining these two results, we can conclude that all honest replicas continuously move through epochs, with each replica initiating a new epoch within Δ time of any other honest replica. □

Lemma 3. *If an honest replica directly commits block B_k in epoch e, then (i) no block different than B_k can be certified in epoch e, and (ii) every honest replica locks on block B_k in epoch e.*

Proof. Suppose an honest replica r directly commits B_k in epoch e at time t (line 32 in Algorithm 2). This means that at time $t-2\Delta$, r received $C_e(B_k)$, locked on it, and started $timeoutCommit(e)$ (lines 18–24 in Algorithm 2). Moreover, replica r forwarded all messages representing $C_e(B_k)$ (lines 17 and 27 in Algorithm 2) so all honest replicas received these messages in Δ time, by time $t - \Delta$.

For part (i), assume for a contradiction that some honest replica p received and voted in epoch e for the block $B_l \neq B_k$. Since every honest replica votes only once, p must have received a proposal for B_l before receiving a proposal message for B_k, at some time $t_1 < t - \Delta$. As a result, p forwards the propose message for B_l at time t_1 (line 17 in Algorithm 2). Replica r will receive this message by time $t_1 + \Delta$, that is, before t. Since these two propose messages lead to a $C_e(\text{EQUIV})$ certificate, p would not commit (lines 9–12 and 15 in Algorithm 3), a contradiction. Therefore, property (i) holds since no honest replica votes for a block different than B_k; otherwise replica r would not commit.

For part (ii), we know by (i) that if replica r directly commits in epoch e, there is not any possible $C_e(B_l) \neq C_e(B_k)$. So, we need to prove that every honest

replica p receives $C_e(B_k)$ before receiving $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$. For a contradiction, assume that p receives $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ before receiving $C_e(B_k)$. This must happen at time $t_1 < t - \Delta$ as p receives $C_e(B_k)$ by time $t - \Delta$. After receiving $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$, p broadcasts them (line 17 in Algorithm 3). So, p broadcasts $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ at time t_1 and r receives them by time $t_1 + \Delta$. Since $t > t_1 + \Delta$ replica r will not commit B_k, a contradiction.

□

Lemma 4. *If B_k is the only certified block in epoch e and $f + 1$ honest replicas lock on block B_k in epoch e ($lockedBlock = B_k$ and $lockedBC = C_e(B_k)$), then in all epochs $e' > e$, they vote only for blocks extending B_k, or they blame the proposer.*

Proof. The proof proceeds by induction on the epoch number.

Base step ($e' = e + 1$) : Let C denote the set of $f + 1$ honest replicas. The replicas in set C do not vote for proposals that do not extend blocks certified in epochs higher than or equal to their $lockedBC$ (line 14 in Algorithm 2). As a result, when $timeoutCertificate(e')$ expires, no block certificate will be formed since no honest replica has voted, causing honest replicas to blame the proposer by sending $\langle \text{BLAME}, e' \rangle_*$ message. Therefore, the lemma holds for the base step since honest replicas vote only for a block if it extends $lockedBlock$.

Induction step ($e' \rightarrow e' + 1$): Assume that no replica in set C has voted for a block not extending B_k until epoch $e' + 1$. We now show that the lemma holds for epoch $e' + 1$. Since replicas from the set C vote for blocks extending B_k or blame the proposer in epochs $e \leq e'' \leq e'$, no block B_l not extending B_k can receive $f + 1$ votes in those epochs. Therefore, for all processes in set C, $lockedBlock = B_{k'}$ and $lockedBC.epoch \geq e$, where $B_{k'} = B_k$ or $B_{k'}$ extends B_k. Assume, for the sake of contradiction, that a process p in set C votes in epoch $e' + 1$ for a block not extending B_k. An honest replica will not vote for a block not extending its $lockedBlock$ (line 14 in Algorithm 2), leading to a contradiction. Hence, the lemma holds for epoch $e' + 1$ as well.

□

Lemma 5. *If an honest replica directly commits block B_k in epoch e, then any block B_l that is certified in epoch $e' > e$ must extend B_k.*

Proof. The proof follows directly from Lemmas 3 and 4. More precisely, if an honest replica directly commits block B_k in epoch e, by Lemma 3, we know that $f + 1$ honest replicas (set C) lock on block B_k in epoch e and B_k is the only certified block in epoch e. Consequently, by Lemma 4, replicas from C vote only for the blocks extending block B_k in epochs $e' > e$. Therefore, no block B_l that

does not extend B_k can collect $f + 1$ votes and thus cannot be certified in any epoch $e' > e$. □

Theorem 2. *(Agreement) No two honest replicas commit different blocks at the same height.*

Proof. Suppose, for the sake of contradiction, that two distinct blocks B_k and B'_k are committed for the same height k. Assume that B_k is committed as a result of B_l being directly committed in epoch e and B'_k is committed as a result of $B_{l'}$ being directly committed in epoch e'. Without loss of generality, assume $l < l'$. Note that all directly committed blocks are certified. This is true because in order to start $timeoutCommit(e)$ for block B_k, a replica needs to receive $C_e(B_k)$ (lines 18–24 in Algorithm 2). By Lemma 5, $B_{l'}$ extends B_l. Therefore, $B_k = B'_k$, which contradicts the assumption that B_k and B'_k are distinct. Hence, no two honest replicas can commit different blocks at the same height.

□

Lemma 6. *If an honest replica r locks on a block B_k in epoch e, no honest replica starts epoch $e+1$ before updating $validBC$ to $C_e(B_l)$, where B_l does not need to be equal to B_k.*

Proof. Assume that an honest replica r locks on a block B_k in epoch e at time t. This implies r receives $C_e(B_k)$ at time t and does not receive $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ before that. Since all messages representing $C_e(B_k)$ are broadcast (lines 17 and 27 in Algorithm 2), all honest replicas receive these messages by time $t + \Delta$.

Suppose for a contradiction that some honest replica p starts the epoch $e+1$ before receiving $C_e(B_k)$ or some other $C_e(B_l)$, in other words at time $t_1 < t + \Delta$. This means it had received $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ and broadcast messages representing them at time $t_1 - 2\Delta$ (line 17 in Algorithm 3). Consequently, replica r receives $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ by time $t_1 - \Delta$ and as $t > t_1 - \Delta$, it does not lock on B_k, a contradiction.

□

Corollary 1. *Every honest replica starts epoch e with $validBC$ that is at least as recent as any certificate any honest replica locks on in any epoch $e' < e$.*

Proof. Suppose that the last epoch in which some honest replica locks on a block is epoch $e' < e$. By Lemma 6, we know that all honest replicas update their $validBC$ to some certificate from the same epoch (e'), before starting epoch $e' + 1$. From this and the fact that no honest replica, in any of the following epochs ($e' < e'' < e$), updates its $validBC$ to an older certificate (lines 25–26 in Algorithm 2), we see that this corollary holds. □

Theorem 3. *(Progress) All honest replicas keep committing new blocks.*

Proof. From Lemmas 1 and 2, we see that replicas proceed through epochs, each epoch having a dedicated leader. If the leader of an epoch is Byzantine and does not propose any block or proposes equivocating blocks, honest replicas will collect $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ and move to the next epoch. Due to the round-robin leader election, there will be epochs with honest leaders.

Consider an epoch e with an honest leader l. Let t be the time when the first honest replica starts epoch e. By Lemma 2, all honest replicas enter epoch e by the time $t + \Delta$. Therefore, by the time $t + \Delta$ at the latest, an honest leader l broadcasts the proposal $\langle \text{PROPOSE}, e, B_k, validBC_l \rangle_l$. All honest replicas receive proposal by time $t + 2\Delta$. Since by the Corollary 1, $validBC_l$ is at least as recent as any $lockedBC$ of any honest replica, all honest replicas vote for the proposal. As a result, all honest replicas receive $C_e(B_k)$ by time $t + 3\Delta$. Since $timeoutCertificate(e) > 3\Delta$, no honest replica will send a $\langle \text{BLAME}, e \rangle_*$ message in epoch e, and $C_e(\text{BLAME})$ cannot be formed. Furthermore, considering that replica l is honest, it does not equivocate, so no $C_e(\text{EQUIV})$ will be formed in epoch e. Consequently, all honest replicas start $timeoutCommit(e)$, and when it expires, they commit B_k and all its ancestors. This scenario will occur in every epoch with an honest leader, ensuring that all honest replicas consistently commit new blocks across all such epochs. □

Theorem 4. *(External validity) Every committed block satisfies the predefined* valid() *predicate.*

Proof. This follows directly from the requirement that every committed block must first be certified (lines 18, 24, and 32 in Algorithm 2). This implies that at least one honest replica accepted the block, meaning that $valid()$ returned true for this block on at least one honest replica (line 11 in Algorithm 2). □

4.3 Debunking synchrony violations

BoundBFT relies on synchrony every time an action is triggered by a timeout. Each timeout is expressed as a multiple of a synchrony bound Δ and is intended to capture a specific exchange of messages between honest replicas. Consequently, a synchrony violation may result in a scenario where the timeout expires before a replica receives an expected message from some honest replica. We refer to this phenomenon as a *timeout violation*.

In this section, we first explore how malicious replicas might attempt to compromise BoundBFT. We then examine the consequences of timeout violations in both the presence and absence of malicious replicas by analyzing each BoundBFT timeout in detail. Finally, we propose a Byzantine protocol informed by these insights.

4.3.1 Byzantine behavior

Listing possible faulty behaviors of Byzantine replicas is unusual since, by definition, a Byzantine replica can behave arbitrarily. However, in the context of leader-based protocols where messages are signed, the scope for deviation is somewhat limited. In the following, we outline the possible faulty behaviors.

Byzantine Leader Behaviors:

- *SILENCE:* The leader does not send a proposal to a subset of replicas, or possibly to any replicas.
- *EQUIVOCATION:* The leader proposes multiple blocks in the same epoch.
- *AMNESIA:* The leader ignores its local knowledge of the blockchain and, instead of extending the blockchain with a new block, proposes an alternative block for a previously committed block.

Non-Leader Byzantine Replica Behaviors:

- *MULTI-VOTE:* A Byzantine replica can vote for any proposal it likes, including multiple proposals in the same epoch.
- *BLAME:* A Byzantine replica can blame the leader by broadcasting a blame message at any point, even if the leader is honest.

In addition, Byzantine replicas can always remain silent or discard messages selectively.

4.3.2 Timeout $timeoutCommit$

The $timeoutCommit$ is the only timeout responsible for BoundBFT's agreement. An honest replica sets this timeout after locking on a block in an epoch (line 24 in Algorithm 2). The replica then forwards the block certificate and waits for this timeout to receive certificates from all other honest replicas.

If $timeoutCommit$ is violated, an honest replica may miss a certificate from some honest replicas. As a result, it may commit block b in epoch e thinking

all honest replicas locked on b, while in reality, some honest replicas received a different certificate and moved to epoch $e+1$ without locking on b. If enough honest replicas did not lock on b, the agreement might be compromised as honest replicas may vote for an alternative block b', create a block certificate, and commit b'.

The likelihood of this situation in the absence of Byzantine replicas is low, as the following conditions must be fulfilled:

1. A blame certificate must be formed in epoch e, meaning a majority of replicas must have blamed the leader in epoch e (i.e., $timeoutCertificate$ was violated in all of these replicas).

2. A majority of replicas did not lock on b in epoch e, receiving a blame certificate before receiving a block certificate for b.

3. The leader of epoch $e+1$ did not receive b's block certificate in epoch e, thus not updating its $validBlock$ and $validBC$ to b and b's certificate (i.e., its $timeoutEpochChange$ was violated in epoch e).

Even though $timeoutCertificate$ and $timeoutEpochChange$ are responsible for BoundBFT's progress, in this scenario, they also play a role in guarding the protocol's agreement.

Byzantine replicas can exploit $timeoutCommit$ violations and potentially compromise BoundBFT's agreement through the following attacks:

- *AMNESIA-ATTACK*: The Byzantine leader ignores the algorithm (line 9 in Algorithm 2) and does not propose a block that extends its *validBlock*. Instead, it proposes an alternative block b for its *validBlock* (i.e., $b.prev = validBlock.prev$). Byzantine replicas vote for this proposal. The agreement can be violated if an honest replica committed $validBlock$ while some honest replicas, due to $timeoutCommit$ violations, did not lock on $validBlock$. As a result, these replicas will vote for block b, and if their votes, together with Byzantine votes, form a majority, block b will be certified and committed. To increase the probability of this scenario, in epochs with an honest leader, Byzantine replicas send votes for the block proposed by the honest leader to one subset of honest replicas to help them form the block certificate faster and commit a block. At the same time, they send BLAME messages to a different subset of honest replicas to help them form a blame certificate before receiving a block certificate.

- *EQUIVOCATION-ATTACK*: The Byzantine leader proposes two distinct proposals in the same epoch, and Byzantine replicas vote for both. The first

proposal and its votes are sent to one subset of honest replicas, and the second proposal and its votes to another subset. As a result, two honest replicas may vote for, receive block certificates, and commit different blocks if their $timeoutCommit$s expire before they learn about the equivocated proposal. In epochs when the leader is honest, Byzantine replicas remain silent and update $validBlock$ and $validBC$ to stay aware of the most recently certified block. This is important so that when they become the leader, Byzantine replicas can generate new blocks that honest replicas will vote for.

4.3.3 Timeout $timeoutCertificate$

Honest replicas initiate this timeout upon starting an epoch (line 2 in Algorithm 3). Its purpose is twofold. First, it ensures that an honest replica does not wait indefinitely for a silent malicious leader. Second, when the leader is honest and proposes a block that all honest replicas will vote for, $timeoutCertificate$ should not expire before all honest replicas receive the proposal and the votes from all other honest replicas. In other words, before they receive a block certificate. Consequently, no honest replicas will blame an honest leader.

If $timeoutCertificate$ is violated, an honest replica will incorrectly blame the honest leader. If a majority of honest replicas blame the leader, a blame certificate can be formed, and the decision might not be reached in the current epoch. However, the block will be committed when the next honest leader proposes a block and other honest replicas receive the block certificate on time.

The creation of a blame certificate is easier in the presence of Byzantine replicas:

- *BLAME-ATTACK*: Byzantine replicas do not vote for the proposal sent by the honest leader. Instead, they broadcast BLAME messages upon starting the epoch with an honest leader. As a result, the blame certificate can be formed if a single honest replica blames the leader. In epochs when the leader is Byzantine, the Byzantine replicas just remain silent.

Apart from unconditionally blaming the leader and hoping that one of the honest replicas will also blame the leader, there is no other way for Byzantine replicas to prevent honest replicas from committing a block in epochs with honest leaders. $timeoutCertificate$ violations may slow down the execution but will not lead to violations in agreement (i.e., when two honest replicas decide on different blocks).

4.3.4 Timeout $timeoutEpochChange$

An honest replica r triggers this timeout when it receives a blame or equivocation certificate as the first certificate in an epoch (line 18 in Algorithm 3). This timeout ensures that if another honest replica receives a block certificate as the first certificate and locks on it in the same epoch, r will receive this block certificate and update its $validBlock$ and $validBC$ before starting the next epoch. Consequently, r starts the next epoch when it receives a block certificate from the current epoch or when $timeoutEpochChange$ expires.

If the *timeoutEpochChange* is violated, an honest replica will not hear about the locked block. Consequently, if the replica is the next epoch leader, it will propose a block that locked replicas will not accept. If the set of remaining honest replicas that vote for a proposed block is less than the majority, the block certificate will not be formed, and a decision will not be reached even though the epoch leader is honest. However, the new block will be committed when one of the locked honest replicas becomes a leader.

Honest replicas rely on this timeout only in epochs when an equivocation or a blame certificate is formed. In the absence of attacks, creating an equivocation certificate is impossible. As a result, an honest replica uses $timeoutEpochChange$ solely in case it receives a blame certificate. This can happen only if $timeoutCertificate$ is violated on a majority of honest replicas, and they blame the honest leader.

With Byzantine replicas, however, both equivocation and blame certificates are possible. Byzantine replicas can exploit $timeoutEpochChange$ violations as follows:

- *EQUIVOCATION-CERTIFICATE-ATTACK*: The Byzantine leader broadcasts a proposal for block b. Then, the Byzantine replicas send votes for block b to one subset of honest replicas to help them create a block certificate and lock on b. At the same time, the Byzantine leader sends a second proposal for block $b' \neq b$ to the other subset of honest replicas. These honest replicas will form an equivocation certificate and start $timeoutEpochChange$. As a result, they will not lock on block b.

- *BLAME-CERTIFICATE-ATTACK*: Similarly, Byzantine replicas impose locking on one subset of honest replicas by sending a proposal and votes for block b. Instead of equivocating, the Byzantine leader remains silent and does not send any proposal to the other subset of honest replicas. Moreover, all Byzantine replicas send blame messages to these replicas. If these replicas did not receive a block certificate before the $timeoutCertificate$

expires,[1] they will blame the leader and, together with blame messages from Byzantine replicas, form a blame certificate and start $timeoutEpochChange$, without locking on block b.

In both attacks, Byzantine replicas remain silent in epochs with an honest leader. By remaining silent, these replicas ensure that if a $timeoutEpochChange$ violation happened and the next honest leader proposes a block that does not extend block b, a block certificate will not be formed and a decision will not be reached.

Similarly to $timeoutCertificate$, violations of $timeoutEpochChange$ can slow down the execution but cannot lead to violations in agreement.

4.3.5 The Byzantine protocol

Algorithm 4 presents the Byzantine replica protocol. Byzantine replicas proceed through epochs in the same way as honest replicas. Namely, if they receive a block certificate, they start the next epoch immediately (lines 23–28 in Algorithm 4), while if they receive a blame or equivocation certificate they wait for $timeoutEpochChange$ before starting the next epoch (lines 29–36 in Algorithm 4). They do this to be synchronized with honest replicas so they can launch the attack at the moment that maximizes the attack's effectiveness. Moreover, a Byzantine replica waits for $timeoutEpochChange$ to update its $validBlock$ and $validBC$ to the most recent values. As a result, when leader in an epoch, a Byzantine replica can propose a valid block (i.e., an invalid block would be easily dismissed by honest replicas).

Algorithms 5 and 6 present the logic for attacks on BoundBFT's agreement and progress, respectively. We empower the attacks by assuming that Byzantine replicas know each other and collude (Section 2.1.1): each Byzantine replica has private keys of all Byzantine replicas. Therefore, a Byzantine replica can sign and send messages on behalf of other Byzantine replicas.

Upon starting an epoch, a Byzantine replica launches a specific attack, which is a parameter of an algorithm:

- *EQUIVOCATION-ATTACK*,
- *AMNESIA-ATTACK*,

[1] Even though the Byzantine replicas do not send the proposal and votes to these replicas, they can receive the forwarded messages from other honest replicas and form a block certificate before $timeoutCertificate$ expires.

Algorithm 4 The Byzantine protocol

```
1: Initialization:
2:   e_p := 0                                                          ▷ the current epoch
3:   validBC_p := nil                                                  ▷ the most recent block certificate the replica is aware of and...
4:   validBlock_p := nil                                               ▷ the block certified by validBC_p
5:   C := getAllHonestReplicas()                                       ▷ the set of honest replicas
6:   f := getNumberOfByzantineReplicas()                               ▷ the number of Byzantine replicas
7:   attackType := getAttackType()                                     ▷ the attack type the Byzantine replica launches
8:   k := getTargetSize()                                              ▷ the size of the two random sets of honest replicas that are under the attack
9: when bootstrapping do StartEpoch(0)                                 ▷ the execution starts in epoch 0
10: Procedure StartEpoch(e) :                                          ▷ upon starting the epoch...
11:   e_p ← e                                                          ▷ the replica sets the current epoch, and...
12:   switch attackType :                                              ▷ invokes the specific attack and pass the necesarry arguments to it
13:     case EQUIVOCATION-ATTACK :
14:       LaunchEquivocationAttack(e, validBlock, validBC, C, k, f)
15:     case AMNESIA-ATTACK :
16:       LaunchAmnesiaAttack(e, validBlock, C, k, f)
17:     case BLAME-ATTACK :
18:       LaunchBlameAttack(e, C, f)
19:     case EQUIVOCATION-CERTIFICATE-ATTACK :
20:       LaunchEquivocationCertificateAttack(e, validBlock, validBC, C, k, f)
21:     case BLAME-CERTIFICATE-ATTACK :
22:       LaunchBlameCertificateAttack(e, validBlock, validBC, C, k, f)
23: when receive ⟨PROPOSE, e, b, BC⟩_l and f + 1 distinct ⟨VOTE, e, id(b)⟩_*   ▷ when replica receives a proposal and f + 1 votes for it...
24:  where e = e_p do                                                  ▷ in the current epoch...
25:   cert ← NewCert from f + 1 ⟨VOTE, e, id(b)⟩_*                     ▷ it forms a block certificate,...
26:   validBC_p ← cert                                                 ▷ updates its validBC_p and validBlock_p...
27:   validBlock_p ← b                                                 ▷ to the most recent block, and...
28:   StartEpoch(e + 1)                                                ▷ starts immidiately the next epoch
29: when receive ⟨PROPOSE, e, b, BC⟩_p and ⟨PROPOSE, e, b′, BC′⟩_p     ▷ upon receiving two proposals...
30:  where e = e_p and p = leader(e) and b ≠ b′ do                      ▷ in the current epoch, from the leader for two distinct blocks...
31:   start timeoutEpochChange(e_p)                                    ▷ the replica triggers timeoutEpochChange
32: when receive f + 1 distinct ⟨BLAME, e⟩_* where e = e_p do           ▷ upon receiving a blame certificate in the current epoch...
33:   start timeoutEpochChange(e_p)                                    ▷ the replica triggers timeoutEpochChange
34: when timeoutEpochChange(e) expires do                              ▷ when timeoutEpochChange expires and...
35:   if e = e_p then                                                  ▷ the replica is stil in epoch e...
36:     StartEpoch(e_p + 1)                                            ▷ the replica starts the next epoch
```

- *BLAME-ATTACK*,
- *EQUIVOCATION-CERTIFICATE-ATTACK*, or
- *BLAME-CERTIFICATE-ATTACK*.

All Byzantine replicas launch the same attack, not only the current epoch leader. This ensures that messages arrive at their destinations as fast as possible. So, if the malicious leader is far from some honest replica, the honest replica will receive attack messages from its closest Byzantine replica. For example, if the attack is *EQUIVOCATION-ATTACK*, each Byzantine replica will generate two proposals and votes for these proposals and send one proposal and its votes to one subset of honest replicas and another proposal and its votes to the other subset of honest replicas.

In attacks where Byzantine replicas divide honest replicas into two subsets and send different messages to them, the subsets are picked randomly. The size

Algorithm 5 Byzantine attacks on BoundBFT's agreement

```
1: Procedure LaunchEquivocationAttack(e, validBlock, validBC, honestReplicas, k, f): ▷ ***EQUIVOCATION-ATTACK***
2:   if isByzantineLeader(e) then ▷ if the epoch leader is a Byzantine replica:
3:     p1, p2 ← generateTwoDifferentProposals(e, validBlock, validBC) ▷ the replica creates two distinct proposals
4:     v1 ← generateVoteMessages(p1, f) ▷ then, the replica generates f VOTE messages for p1, and...
5:     v2 ← generateVoteMessages(p2, f) ▷ f VOTE messages for p2, one for each Byzantine replica
6:     set1, set2 ← getTwoRandomSets(e, k, honestReplicas) ▷ lastly, the replica divide honest replicas in two random sets of size k
7:     send p1 and v1 to set1 ▷ then, it sends the first proposal and votes for it to the first set, set1,
8:     send p2 and v2 to set2 ▷ while, sending the second proposal and its votes to the second set, set2
9: Procedure LaunchAmnesiaAttack(e, validBlock, honestReplicas, k, f): ▷ ***AMNESIA-ATTACK***
10:  if isByzantineLeader(e) then ▷ if the epoch leader is a Byzantine replica:
11:    p ← generateAlternativeProposal(e, validBlock) ▷ the replica generates an alternative proposal for validBlock
12:    v ← generateVoteMessages(p, f) ▷ then, the replica generates f VOTE messages for p, one for each Byzantine replica
13:    send p and v to honestReplicas ▷ the replica sends the proposal and votes for it to the all honest replicas
14:  else ▷ else, if the leader is an honest replica:
15:    set1, set2 ← getTwoRandomSets(e, k, honestReplicas) ▷ the replica divides honest replicas, in two random sets of size k
16:    upon receiving p = ⟨PROPOSE, e, b, BC⟩l ▷ then, when it receives the proposal...
17:     where l = leader(e) do ▷ from epoch leader...
18:      v ← generateVoteMessages(p, f) ▷ it generates f VOTE messages for received proposal and...
19:      b ← generateBlameMessages(e, f) ▷ f BLAME messages for epoch e, one for each Byzantine replica
20:      send v to set1 ▷ then, it sends votes to one subset of honest replicas, set1,...
21:      send b to set2 ▷ while sending blames to the second subset of honest replicas, set2
```

Algorithm 6 Byzantine attacks on BoundBFT's progress

```
1: Procedure LaunchBlameAttack(e, honestReplicas, f): ▷ ***BLAME-ATTACK***
2:   if isHonestLeader(e) then ▷ if the epoch leader is honest:
3:     b ← generateBlameMessages(e, f) ▷ the replica generates f BLAME messages, one for each Byzantine replica, and...
4:     send b to honestReplicas ▷ send them to all honest replicas
5: Procedure LaunchEquivocationCertificateAttack(e, validBlock, validBC, k, ▷ ***EQUIVOCATION-CERTIFICATE-ATTACK***
6:  honestReplicas, f):
7:   if isByzantineLeader(e) then ▷ if the epoch leader is Byzantine:
8:     p1, p2 ← generateTwoDifferentProposals(e, validBlock, validBC) ▷ the replica generates two different proposals, and...
9:     v1 ← generateVoteMessages(p1, f) ▷ f VOTE messages only for p1, one for each Byzantine replica
10:    set1, set2 ← getTwoRandomSets(e, k, honestReplicas) ▷ then, the replica divides honest replicas in two random sets of size k
11:    send p1 and v1 to set1 ▷ finally, it sends the first proposal and votes for it to the first set of honest replicas, set1,...
12:    send p1 and p2 to set2 ▷ while sending both proposals to the second set of honest replicas, set2
13: Procedure LaunchBlameCertificateAttack(e, validBlock, validBC, k, ▷ ***BLAME-CERTIFICATE-ATTACK***
14:  honestReplicas, f):
15:   if isByzantineLeader(e) then ▷ if the epoch leader is Byzantine:
16:     p ← generateNewProposal(e, validBlock) ▷ the replica generates new proposal for epoch e,...
17:     v ← generateVoteMessages(p, f) ▷ f VOTE messages for proposal, and...
18:     b ← generateBlameMessages(e, f) ▷ f BLAME messages, one for each Byzantine replica
19:     set1, set2 ← getTwoRandomSets(e, k, honestReplicas) ▷ then, replica divides honest replicas in two random sets of size k
20:     send p and v to set1 ▷ finally, it sends the proposal and votes for it to the first set of honest replicas, set1,...
21:     send b to set2 ▷ while sending blame messages to the second set of honest replicas, set2
```

of these subsets is a parameter of the algorithm, defined by k. If k is set to 2, function $getTwoRandomSets(e,k,set)$ will return two different subsets, each containing two random elements. Byzantine replicas ensure they have the same subsets by using the current epoch number as a random number generator seed.

4.4 Experimental evaluation

In this section, we first introduce the evaluation setup (Section 4.4.1). Next, we explain how we determined the synchronous bounds for BoundBFT (Section 4.4.2) and compare the performance of BoundBFT with state-of-the-art synchronous and partially synchronous consensus algorithms (Section 4.4.3). We conclude with a summary of the main findings (Section 4.4.5).

4.4.1 Evaluation setup

This section explains the choice of protocols we compare BoundBFT to. Then, it provides details about the experimental environment and implementation.

Methodology

We compare BoundBFT to state-of-the-art leader-rotating Byzantine consensus protocols in the synchronous and partially synchronous system models (see Table 4.2). Protocols designed to allow frequent leader rotation provide better blockchain fairness and censorship resistance than protocols where the leader is not expected to change often (e.g., PBFT) [5]. In leader-rotating protocols, a new replica is elected leader when the protocol changes epoch (or round or view) as part of the normal execution, not just in case of a leader failure. This way, every replica has a chance to be the leader, propose a new block, and receive a reward for it. Moreover, a Byzantine leader that censors transactions by not including them in proposed blocks has little impact as leaders change frequently.

	System model	Resilience	Pipelining	Optimistic responsiveness
HotStuff-2 [82]	partially synchronous	$f < n/3$	yes	yes
Tendermint [20]	partially synchronous	$f < n/3$	no	yes
Sync HotStuff [5]	synchronous	$f < n/2$	yes	yes
BoundBFT	synchronous	$f < n/2$	yes	yes

Table 4.2. Protocols in our evaluation and their main characteristics.

Additionally, we consider protocols that allow optimistic responsiveness [82], meaning that in good cases, where we have a sequence of honest leaders, protocols change leaders responsively, waiting for real network delays only. Lastly, we consider protocols that use pipelining [119], an optimization where replicas start working on the next block only after receiving the certificate for the previous

block, i.e., they do not wait for the previous block to be committed. In doing so, these protocols can order multiple blocks in parallel, and commit latencies have a lesser impact on throughput. We include synchronous and partially synchronous representative protocols that benefit from this optimization.

In the synchronous model, we consider a version of Sync HotStuff that supports responsive leader rotation, pipelining, and has optimal latency [3; 5]. In the partially synchronous model, we choose Tendermint [20] and HotStuff-2 [82], the most recent protocol of the HotStuff family [119], with pipelining. Tendermint provides optimistic responsiveness, and due to its quadratic communication, it needs only three communication steps to commit a value. As such, Tendermint serves as a good baseline for latency. However, Tendermint does not implement pipelining. Consequently, we use HotStuff-2 as a baseline for throughput. HotStuff-2 implements pipelining and optimistic responsiveness but has higher latency than Tendermint since it requires five linear communication steps to commit a value.

Environment

We conducted our experiments in a cluster with emulated wide-area latencies between 6 AWS zones (see Table 4.1). Latencies between cluster nodes were configured using the Linux Traffic Control kernel module [62]. The emulated WAN provided an affordable approximation of the AWS environment since our evaluation required hundreds of hours of experiments. The cluster contains 60 machines divided in two groups: (i) EPYC Zen 2 with two 16-Core AMD EPYC 2881 MHz and 32GB of RAM, and (ii) HP SE1102 with two Quad-Core Intel Xeon 2.5GHz and 8GB of RAM.

Implementation

To provide a fair comparison, we implemented BoundBFT and all competing protocols (see Table 4.2) in Go. The implementations use SHA256 hashes and Ed25519 64-byte digital signatures. We rely on libp2p [78] to establish and maintain communication channels between pairs of replicas.

To generate a load in our experiments, we equip every replica with a built-in client that generates transactions in advance and stores them in a local pool. When a replica is a leader in an epoch, it takes transactions from the pool and forms a block. The block size defines the number of transactions taken from the pool. This design leaves the mempool (i.e., the part of a blockchain responsible for propagating client transactions across the system) out of the discussion as

different systems may implement it in different ways. Consequently, the latencies we report here represent consensus latencies (i.e., the time the leader of an epoch needs to commit a block). Throughput is computed as the rate of committed blocks per time unit.

4.4.2 BoundBFT's synchrony bound

In this section, we experimentally determine the value for BoundBFT's synchrony bound Δ. Instead of defining a value that ensures, with high probability, that synchronous system assumptions hold, we ran BoundBFT in the presence of malicious replicas to determine a Δ that provides sufficient confidence that BoundBFT's correctness will not be compromised. We implemented all proposed attacks (see Algorithms 4–6).

We then ran BoundBFT in our cluster with f Byzantine replicas. As a starting point, we set Δ to 1250 ms (99.99%), the synchronous bound from [80], and gradually decreased it until we started to observe BoundBFT's agreement and progress violations. Tables 4.3 and 4.4 show the complete data from these experiments. We varied the block size, number of Byzantine replicas (f), and the size of a partition (k) for attacks that partition honest replicas into two subsets. The main takeaways from our experiments are as follows.

When there is a single ($f = 1$) or no ($f = 0$) Byzantine replicas in the system, we did not observe any agreement violations, even if we set Δ as low as 50 ms—the average latency between 80% of replicas in our system is higher than 50 ms. This shows that agreement violations are highly unlikely if the number of Byzantine replicas is low, even if many messages violate synchronous bounds.

However, the situation changes as the number of Byzantine replicas increases. When the number of Byzantine replicas is $f = 19$ (i.e., the maximum number of Byzantine replicas partially synchronous protocols can tolerate), agreement violations were observed only when we lowered Δ to 50 ms. Even then, with $\Delta = 50$ ms, BoundBFT's agreement was violated in less than 10% of epochs in which the attack was launched.

To prevent agreement violations when the number of Byzantine replicas is $f = 29$ (i.e., the maximum BoundBFT can tolerate), we needed to increase Δ to 150 ms and 300 ms for 1KB and 32KB block sizes, respectively. This makes sense since to create a block certificate, an honest replica needs to receive a vote from itself, Byzantine replicas, and one other honest replica. Importantly, even in this case, the resulting Δ was 8 and 4 times lower than the initial one. Notice that when $f = 29$, partially synchronous protocols will halt if Byzantine replicas remain silent.

N = 60, 10 minutes experiments, block size 1KB												
Δ (ms)	No attack											
	Safety						Liveness					
1250	0%						0%					
600	0%						0%					
300	0%						0%					
150	0%						0%					
100	0%						0%					
50	0%						65%					
Δ (ms)	Equivocation attack											
	f=29				f=19				f=1			
	tSize=1		tSize=all		tSize=1		tSize=all		tSize=1		tSize=all	
	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	1%	0%	3%	0%	0%	0%	0%	0%	0%	0%	0%
100	5%	36%	6%	6%	0%	5%	0%	0%	0%	0%	0%	0%
50	19%	80%	31%	60%	2%	73%	8%	54%	0%	65%	0%	65%
Δ (ms)	Amnesia attack											
	f=29				f=19				f=1			
	tSize=1		tSize=all		tSize=1		tSize=all		tSize=1		tSize=all	
	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
100	4%	19%	0%	18%	0%	14%	0%	8%	0%	0%	0%	0%
50	26%	90%	0%	80%	9%	81%	0%	97%	0%	66%	0%	67%
Δ (ms)	Blame attack											
	f=29				f=19				f=1			
	Safety		Liveness		Safety		Liveness		Safety		Liveness	
1250	0%		0%		0%		0%		0%		0%	
600	0%		0%		0%		0%		0%		0%	
300	0%		0%		0%		0%		0%		0%	
150	0%		0%		0%		0%		0%		0%	
100	0%		84%		0%		46%		0%		0%	
50	0%		100%		0%		100%		0%		67%	
Δ (ms)	Equivocation certificate attack											
	f=29				f=19				f=1			
	tSize=1		tSize=all		tSize=1		tSize=all		tSize=1		tSize=all	
	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
100	0%	16%	0%	12%	0%	17%	0%	12%	0%	0%	0%	0%
50	2%	88%	0%	90%	0%	86%	0%	80%	0%	63%	0%	66%
Δ (ms)	Blame certificate attack											
	f=29				f=19				f=1			
	tSize=1		tSize=all		tSize=1		tSize=all		tSize=1		tSize=all	
	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
100	0%	21%	0%	25%	0%	15%	0%	17%	0%	0%	0%	0%
50	2%	90%	0%	92%	0%	92%	0%	99%	0%	63%	0%	64%

Table 4.3. Percentage of Agreement and Progress violations when running BoundBFT under different attacks while using different values as its Δ. The table shows data for the setup of 60 replicas, 1KB block size and different number of Byzantine replicas (f). Additionally, for attacks that parition honest replicas in two subsets, it shows results when Byzantine replicas divide them in two minimal subsets ($k = k_{min} = 1$) and two maximal subsets ($k = k_{max} = n - f/2$).

N = 60, 10 minutes experiments, block size 32KB												
Δ (ms)	**No attack**											
	Safety						Liveness					
1250	0%						0%					
600	0%						0%					
300	0%						0%					
150	0%						0%					
100	0%						0%					
50	0%						66%					
Δ (ms)	**Equivocation attack**											
	f=29				f=19				f=1			
	tSize=1		tSize=all		tSize=1		tSize=all		tSize=1		tSize=all	
	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%
150	2%	0%	0%	15%	0%	0%	0%	0%	0%	0%	0%	0%
100	5%	34%	0%	67%	0%	13%	0%	9%	0%	0%	0%	0%
50	27%	91%	5%	85%	1%	94%	0%	97%	0%	66%	0%	66%
Δ (ms)	**Amnesia attack**											
	f=29				f=19				f=1			
	tSize=1		tSize=all		tSize=1		tSize=all		tSize=1		tSize=all	
	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	2%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	3%	26%	1%	10%	0%	1%	0%	0%	0%	0%	0%	0%
100	6%	42%	0%	44%	0%	22%	0%	0%	0%	0%	0%	0%
50	27%	81%	10%	88%	9%	74%	0%	37%	0%	67%	0%	67%
Δ (ms)	**Blame attack**											
	f=29				f=19				f=1			
	Safety		Liveness		Safety		Liveness		Safety		Liveness	
1250	0%		0%		0%		0%		0%		0%	
600	0%		0%		0%		0%		0%		0%	
300	0%		0%		0%		0%		0%		0%	
150	0%		25%		0%		0%		0%		0%	
100	0%		85%		0%		43%		0%		0%	
50	0%		100%		0%		99%		0%		79%	
Δ (ms)	**Equivocation certificate attack**											
	f=29				f=19				f=1			
	tSize=1		tSize=all		tSize=1		tSize=all		tSize=1		tSize=all	
	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	2%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	0%	0%	18%	0%	0%	0%	0%	0%	0%	0%	0%
100	0%	31%	0%	61%	0%	13%	0%	17%	0%	0%	0%	0%
50	3%	86%	0%	83%	0%	95%	0%	95%	0%	67%	0%	67%
Δ (ms)	**Blame certificate attack**											
	f=29				f=19				f=1			
	tSize=1		tSize=all		tSize=1		tSize=all		tSize=1		tSize=all	
	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness
1250	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
600	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
300	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
150	0%	2%	0%	3%	0%	0%	0%	0%	0%	0%	0%	0%
100	0%	30%	0%	32%	0%	17%	0%	12%	0%	0%	0%	0%
50	1%	90%	0%	96%	0%	95%	0%	99%	0%	66%	0%	66%

Table 4.4. Percentage of Agreement and Progress violations when running BoundBFT under different attacks while using different values as its Δ. The table shows data for the setup of 60 replicas, 32KB block size and different number of Byzantine replicas (f). Additionally, for attacks that parition honest replicas in two subsets, it shows results when Byzantine replicas divide them in two minimal subsets ($k = k_{min} = 1$) and two maximal subsets ($k = k_{max} = n - f/2$).

Table 4.5 summarizes the results from Tables 4.3 and 4.4, showing the Δs that resulted in no agreement violations and less than 5% of progress violations for all considered setups. Specifically, no two honest replicas committed different blocks for the same height, and in less than 5% of epochs with an honest leader, some honest replicas did not commit a new block. The progress violations can also be lowered to 0%, but this would require a slight increase in the chosen Δ. We believe this is not necessary since progress violations can only lead to a slight decrease in performance and do not affect agreement.

Attack type	1KB						32KB					
	f=29		f=19		f=1		f=29		f=19		f=1	
	k_{min}	k_{max}	k_{min}	k_{max}	k_{min}	k_{max}	k_{min}	k_{max}	k_{min}	k_{max}	k_{min}	k_{max}
EQUIVOCATION	150	150	100	100	100	100	150	300	150	150	100	100
AMNESIA	150	150	150	150	100	100	300	300	150	150	100	100
EQUIVOCATION-CERTIFICATE	150	150	150	150	100	100	300	300	150	150	100	100
BLAME-CERTIFICATE	150	150	150	150	100	100	150	150	150	150	100	100
BLAME	150		150		100		300		150		100	
NO ATTACK	100						100					

Table 4.5. The Δ in ms BoundBFT must adopt to achieve 0% of Agreement and $< 5\%$ of Progress violations under a specific attack. The table shows data for the setup of 60 replicas, 1KB and 32KB block sizes and different number of Byzantine replicas (f). Additionally, for attacks that partition honest replicas in two subsets, it shows results when Byzantine replicas divide honest replicas into the two smallest subsets ($k = k_{min} = 1$) and the two largest subsets ($k = k_{max} = n - f/2$).

4.4.3 Performance

In this section, we compare BoundBFT to state-of-the-art synchronous and partially synchronous protocols (Table 4.2). Every point in the graphs is an average of 3 runs. We ran each experiment for 5 minutes.

Latency

BoundBFT and Sync HotStuff wait for 2Δ (BoundBFT's $timeoutCo\text{-}mmit(e)$) before committing a block in epoch e. Consequently, their latency is directly affected by the chosen synchrony bound. For BoundBFT, we adopt the synchrony bound based on the experiments from the previous section (see Table 4.5). Conversely, for Sync HotStuff, we use the Δ from [80].[2]

[2]Using different bounds for Sync HotStuff would require a similar analysis and evaluation as presented in Sections 4.3 and 4.4.2, which is beyond the scope of this work.

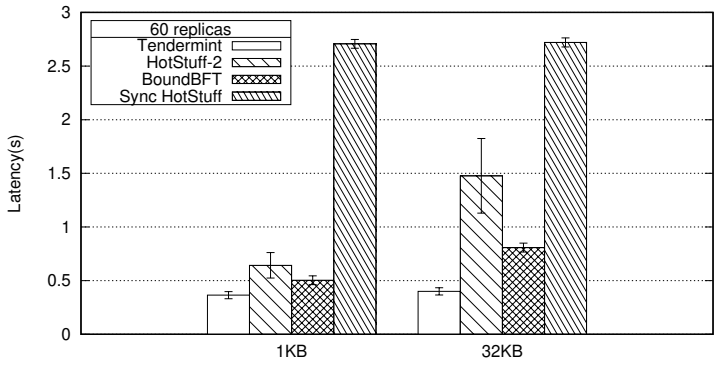

Figure 4.2. Latency comparison for all protocols for 1 KB and 32 KB block sizes in a system with 60 replicas.

We measured latencies in a system with 60 replicas with 1KB and 32KB block sizes. Figure 4.2 shows the average latency computed by epoch leaders. First, we notice the significant impact of BoundBFT's synchrony bound on latency. Namely, BoundBFT achieves 5.4× and 3.4× lower latency than Sync HotStuff with 1KB and 32KB block sizes, respectively. Second, BoundBFT's latency is in between the latencies of partially synchronous protocols. It is 1.3× and 1.8× lower than HotStuff-2's latency and 1.4× and 2× higher than Tendermint's for small and large blocks, respectively. HotStuff-2 has higher latency due to its linear communication pattern, which requires five communication steps, while Tendermint has quadratic communication and commits a block in only three communication steps. We can also see that HotStuff-2 has the most significant standard deviation; we attribute this to the lower redundancy due to its linear communication pattern.

Throughput

BoundBFT and Sync HotStuff [3; 5] use *pipelining* to limit the impact of Δ on throughput, which allows the leader to propose a block B_{k+1} that extends block B_k after receiving $C_e(B_k)$, i.e., before committing block B_k. This way, protocols can order multiple blocks in parallel, and the throughput is unaffected by Δ. This technique was initially introduced in HotStuff [119], and we implemented a pipelined version of HotStuff-2. Adapting pipelining to Tendermint is more complex and out of the scope of this work. Moreover, notice that we compare BoundBFT to a pipelined partially synchronous protocol, HotStuff-2.

We evaluated throughput in a system with 60 replicas and various block sizes (see Figure 4.3). BoundBFT and Sync HotStuff have similar throughout, as they

both start ordering the next block after receiving a certificate for the previous block. Moreover, they outperform partially synchronous protocols for all block sizes considered, reaching throughput more than 2× higher than Tendermint's for all block sizes. The reason behind this is that Tendermint does not use pipelining. They also perform better, from 1.4× to 3×, than HotStuff-2, a partially synchronous protocol with pipelining. This is because even though both protocols start ordering the next block after collecting a certificate for the previous block, the certificate in HotStuff-2 requires votes from a two-third majority of replicas, while in BoundBFT the votes from the majority are enough.

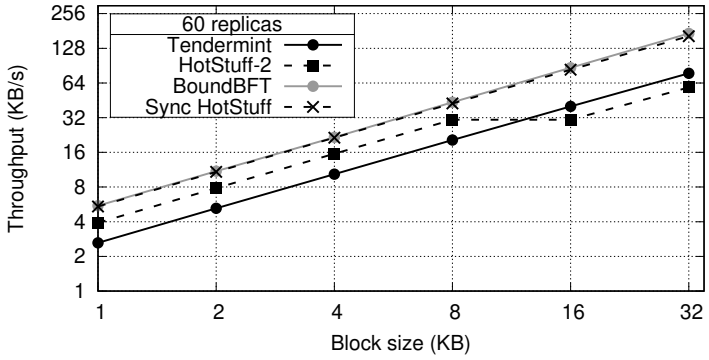

Figure 4.3. Throughput comparison of all protocols in a system with 60 replicas with varying block sizes.

4.4.4 Experiments with large blocks

Previous results (Section 4.4.2) have demonstrated that running BoundBFT with a 32 KB block size necessitates an increased Δ compared to a 1 KB block size. To further explore this relationship, we conducted additional experiments where we increased the block size to 128 KB.

We did not run all possible scenarios as in Section 4.4.2. Instead, we focused on a scenario with no Byzantine replicas and the most efficient attack scenario, the equivocation attack.

The results of these experiments are presented in Table 4.6. As anticipated, the data clearly show that larger block sizes require larger Δ values to maintain BoundBFT's correctness. Specifically, in a scenario with 29 Byzantine replicas, we observed safety violations even when we ran BoundBFT with a conservative Δ from [80] (1250 ms). When we lowered the number of Byzantine replicas, the results improved; we needed to lower the Δ to 50 ms to observe safety violations

in executions with 19 Byzantine replicas.

Additionally, in executions with one or no Byzantine replicas, we did not see any safety violations, similar to previous observations (see Tables 4.3 and 4.4). However, we observed that the percentage of liveness violations is generally much higher than in experiments with smaller blocks. For instance, in executions with no attack, we started to see liveness violations with a Δ of 150 ms, whereas before, we needed to lower the Δ to 50 ms to observe such violations.

N = 60, 10 minutes experiments, block size 128KB												
Δ (ms)	No attack											
	Safety						Liveness					
1250	0%						0%					
600	0%						0%					
300	0%						0%					
150	0%						2%					
100	0%						22%					
50	0%						89%					
Δ (ms)	Equivocation attack											
	f=29				f=19				f=1			
	tSize=1		tSize=all		tSize=1		tSize=all		tSize=1		tSize=all	
	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness	Safety	Liveness
1250	1%	0%	0%	3%	0%	0%	0%	0%	0%	0%	0%	0%
600	6%	0%	0%	6%	0%	0%	0%	0%	0%	0%	0%	0%
300	7%	1%	1%	25%	0%	0%	0%	0%	0%	0%	0%	0%
150	14%	29%	2%	37%	0%	1%	0%	20%	0%	0%	0%	1%
100	33%	64%	12%	32%	0%	48%	0%	55%	0%	22%	0%	23%
50	61%	56%	26%	35%	3%	77%	0%	48%	0%	90%	0%	89%

Table 4.6. Percentage of Agreement and Progress violations when running BoundBFT under equivocation attack and under no attack while using different values as its Δ. The table shows data for the setup of 60 replicas, 128KB block size and different number of Byzantine replicas (f). Additionally, for equivocation attack, it shows results when Byzantine replicas divide them in two minimal subsets ($k = k_{min} = 1$) and two maximal subsets ($k = k_{max} = n - f/2$).

4.4.5 Summary

In this section, we summarize the main takeaways of our evaluation.

- The probability that synchrony violations alone compromise BoundBFT's agreement is deficient, i.e., we did not observe any agreement violations in our experiments when the number of Byzantine replicas was 0 and 1. The agreement can be violated if synchrony violations are combined with a significant number of colluded Byzantine replicas (minority or one-third) that launch the attack.

- BoundBFT can use a Δ that is 4× to 8× smaller than the conservative 99.99% Δ [80], allowing it to improve latency from $\approx$ 3.4× to 5.4×. Moreover, BoundBFT's Δ is big enough to ensure correctness with high probability when the system is under attack. We believe this approach can be used for other synchronous protocols (e.g., Sync HotStuff).
- In addition to higher resilience and availability, BoundBFT achieves from 1.4× to 3× higher throughput and comparable latency to partially synchronous protocols.
- Increasing block size necessitates a larger Δ to maintain BoundBFT's correctness. Conservative Δ based on pings [80] was not big enough to prevent safety violations in executions with a 128 KB block size.

4.5 Conclusion

In this chapter, we introduced a novel approach to determining the synchronous bound Δ. Rather than ensuring all messages are received within synchronous bounds with high probability, we analyzed protocol semantics and designed Byzantine attacks that could potentially cause correctness violations in case of synchrony violations. We then implemented these attacks to determine a Δ that renders them ineffective.

As a showcase, we designed BoundBFT, a new BFT synchronous consensus protocol, and demonstrated experimentally that BoundBFT can use a less conservative synchrony bound without compromising correctness under attacks. Due to the lower synchrony bound, BoundBFT outperforms traditional synchronous consensus protocols. Additionally, it achieves similar latency and better throughput than partially synchronous protocols, offering higher resilience.

Our experiments also indicate that larger block sizes require a larger Δ to maintain BoundBFT's correctness. Specifically, with a 128 KB block size, we observed safety violations even with the conservative 99.99% Δ from [80]. In the following chapter, we will address these challenges and present a solution to manage large block sizes effectively.

Chapter 5

AlterBFT: Fast Synchronous BFT Consensus

5.1 Introduction

In the previous chapter, we identified that large block sizes necessitate large Δ's to maintain the correctness of BoundBFT, a synchronous consensus protocol. To better understand the underlying reasons and the nature of synchrony bounds, we conducted a three-month empirical study on message delays in a geographically distributed system.

Our findings reveal that small messages have significantly lower latency than large messages (e.g., the 99.99% percentile latency is around 300 milliseconds for 4 KB messages and 6 seconds for 1 MB messages). Moreover, small messages exhibit more stable behavior than large messages (e.g., jitter is around 10 milliseconds for 4 KB messages and 390 milliseconds for 1 MB messages).

The key insight from this study is to design protocols whose safety relies solely on small messages, similar to synchronous protocols, while progress depends on the eventual bounds of large messages, akin to partially synchronous protocols. This approach led to the development of AlterBFT, a protocol that achieves performance comparable to partially synchronous protocols, and demonstrates better resilience, tolerating up to $f < n/2$ malicious replicas.

We start off by introducing a new system model, the *hybrid synchronous system model*. The new model captures the timely delivery of "small" messages (i.e., up to 4 KB) and the eventual timely delivery of "large" messages, as in the partially synchronous system model. We then present AlterBFT, a new rotating-leader Byzantine fault-tolerant (BFT) consensus protocol for the hybrid synchronous model. AlterBFT's safety relies on the timely delivery of small messages, whose

time bounds are significantly lower than those of large messages due to lower latency and jitter. We also extend AlterBFT with an optimistic fast path [4; 10; 52; 68]. As a result, AlterBFT achieves responsive latency, which depends on real network delays only, and commits a block in two real network delays in the absence of failures.

We have implemented AlterBFT and compared it to state-of-the-art synchronous and partially synchronous protocols. Experimental evaluation in a geographically distributed environment shows that AlterBFT improves the latency of synchronous protocols from 1.5× to 14.9×, achieving latency comparable to partially synchronous protocols. Furthermore, AlterBFT achieves similar throughput as synchronous protocol, consistently higher than considered partially synchronous protocols, from 1.3× to 7.2×. Lastly, AlterBFT tolerates the same number of failures $f < n/2$ as synchronous protocols, an improvement over partially synchronous protocols, where $f < n/3$.

5.1.1 Outline

The remainder of the chapter is structured as follows. Section 5.2 motivates and defines the new system model. Section 5.3 presents AlterBFT. Section 5.4 experimentally evaluates AlterBFT's performance in a geographically distributed environment and compares AlterBFT to state-of-the-art synchronous and partially synchronous protocols. Section 5.5 concludes the chapter.

5.2 Hybrid synchronous system model

In this section, we motivate our approach based on empirical data observed in a public geographically distributed system and introduce a novel timing model that captures the behavior revealed by the gathered data.

5.2.1 Motivation

In the synchronous system model, messages exchanged between non-faulty replicas must arrive within a predetermined time limit Δ, which serves as the basis for setting timeouts. Accurately understanding communication latency is crucial for determining Δ and establishing realistic timeout values. In this section, we present the findings of an experimental assessment of round-trip latency between five AWS regions (i.e., N. Virginia, S. Paulo, Stockholm, Singapore, and Sydney).

Figure 5.1 shows the root mean square (RMS) jitter in milliseconds for different message sizes. The results show that jitter significantly depends on message size: while small messages (i.e., up to 4KB) experience low jitter, larger messages are subject to significant delay variation, a trend that increases with message size. With 4 KB messages, the jitter for regions 0–3 and 1–4 is 26 ms and 24 ms, respectively. These are the largest values for 4 KB messages across all regions, and represent 22% and 15% of the average communication latency between these regions. Smaller messages have significantly lower jitter across all regions considered.

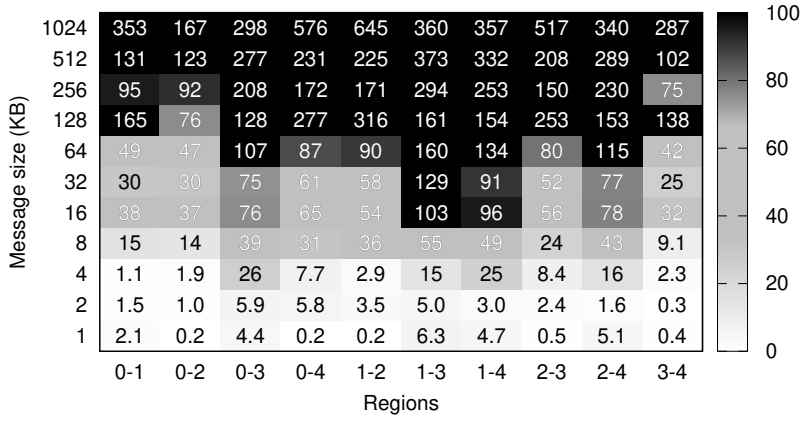

Figure 5.1. Jitter in milliseconds across AWS regions for different message sizes, where 0:N. Virginia (AWS), 1:S. Paulo (AWS), 2:Stockholm (AWS), 3:Singapore (AWS), and 4:Sydney (AWS).

Figure 5.2 presents actual message delays between replicas in different AWS regions. As with jitter, message delays increase with message size, a pattern that holds for all regions considered. Namely, the average message delay for 4KB, 128KB, and 1MB messages is 166ms, 795ms, and 1543ms, respectively. Additionally, to investigate if low and stable latency of small messages is constrained solely to AWS servers, we collected the following: (a) message delays between servers of a different provider, DigitalOcean (Figure 5.3), and (b) message delays between different AWS and DigitalOcean servers (Figures 5.4–5.6). The results confirm that our key observation also holds within servers of a different provider and across providers. Table 5.1 shows the server numbers, their locations, and their providers.

The data presented in Figure 5.1 is based on experiments conducted during 24 hours. This data suggests that large messages have greater and more variable delays than small messages. Table 5.2 shows that the delay of small messages remains low and stable during longer periods of time. More precisely,

Server #	Location	Provider
0	North Virginia	AWS
1	Sao Paulo	AWS
2	Stockholm	AWS
3	Singapore	AWS
4	Sydney	AWS
5	New York	DigitalOcean
6	Toronto	DigitalOcean
7	Frankfurt	DigitalOcean
8	Singapore	DigitalOcean
9	Sydney	DigitalOcean

Table 5.1. The server numbers, their locations, and their providers.

	S. Paulo				Stockholm				Singapore				Sydney			
	99.99%		Jitter		99.99%		Jitter		99.99%		Jitter		99.99%		Jitter	
	1d	3m	1d	3m	1d	3m	1d	3m	1d	3m	1d	3m	1d	3m	1d	3m
N. Virginia	104	101	1.06	3.41	103	107	1.89	4.86	231	238	25.50	13.49	189	166	7.67	3.00
S. Paulo					204	157	2.87	6.94	318	325	14.82	18.86	296	306	24.62	15.64
Stockholm									170	155	8.35	5.28	276	294	16.20	18.14
Singapore													125	125	2.27	2.42

Table 5.2. Message delays (99.99% percentile) and jitter between five distant AWS regions during one day (1d) and three months (3m); all values in milliseconds and 4 KB messages.

we conducted experiments with 4 KB messages over three months and compared the three-month to the one-day data latency (i.e., 99.99% percentile) and jitter. The delays are relatively stable between all considered AWS regions. The most significant increase is 6.5%, observed between Stockholm and Sydney. Most importantly, the largest 99.99% percentile value is relatively small (325ms).

Our empirical evaluation supports that communication delays for small messages are relatively consistent, as their processing and transmission times exhibit minimal variation. This is because small messages are typically transmitted in a single packet, enabling efficient routing and no fragmentation. In contrast, large messages are segmented into multiple packets, potentially traversing different network paths and encountering varying levels of congestion and latency along the way. This inherent heterogeneity in packet delivery introduces variability in the latency of large messages, as the delayed arrival of individual packets can significantly impact their delivery time. Furthermore, transport protocols introduce overhead and delays when large messages are transmitted. For instance, TCP's flow control and congestion control mechanisms aim to ensure reliable and efficient packet delivery, but they can also introduce additional delay and jitter, particularly in large messages.

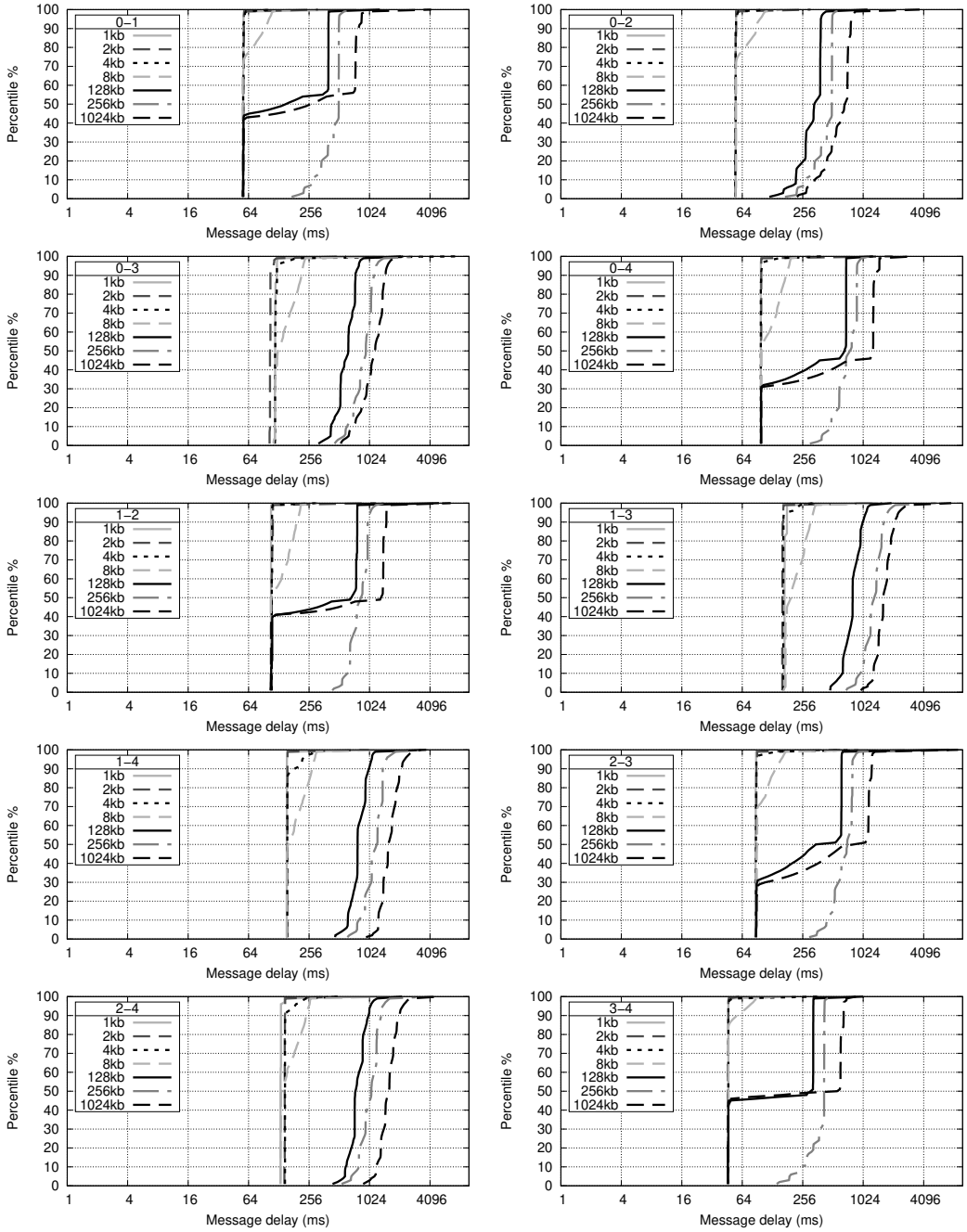

Figure 5.2. Communication delays with various message sizes between different AWS regions (x-axe is in log scale with the base 2).

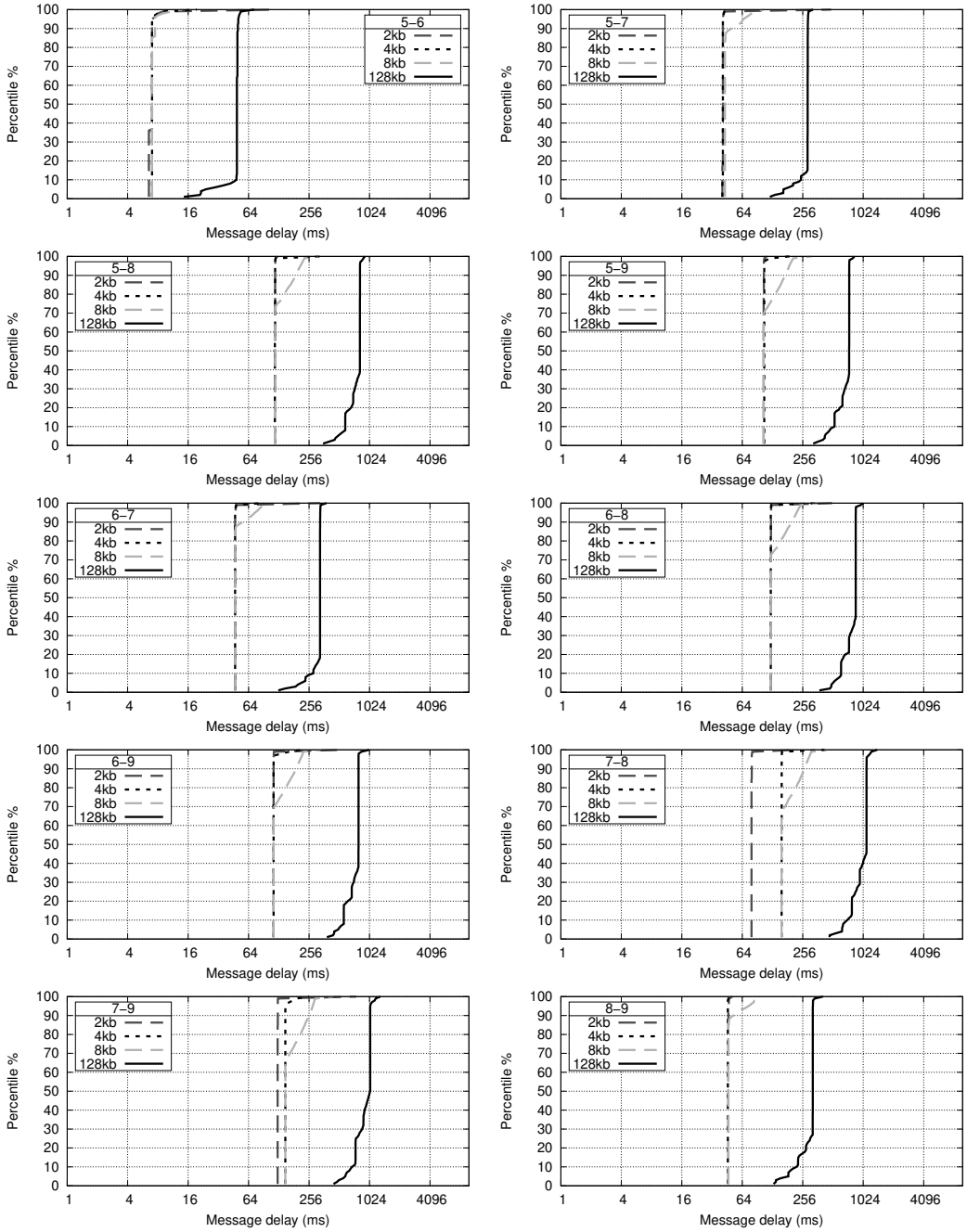

Figure 5.3. Communication delays with various message sizes between different DigitalOcean regions (x-axe is in log scale with the base 2).

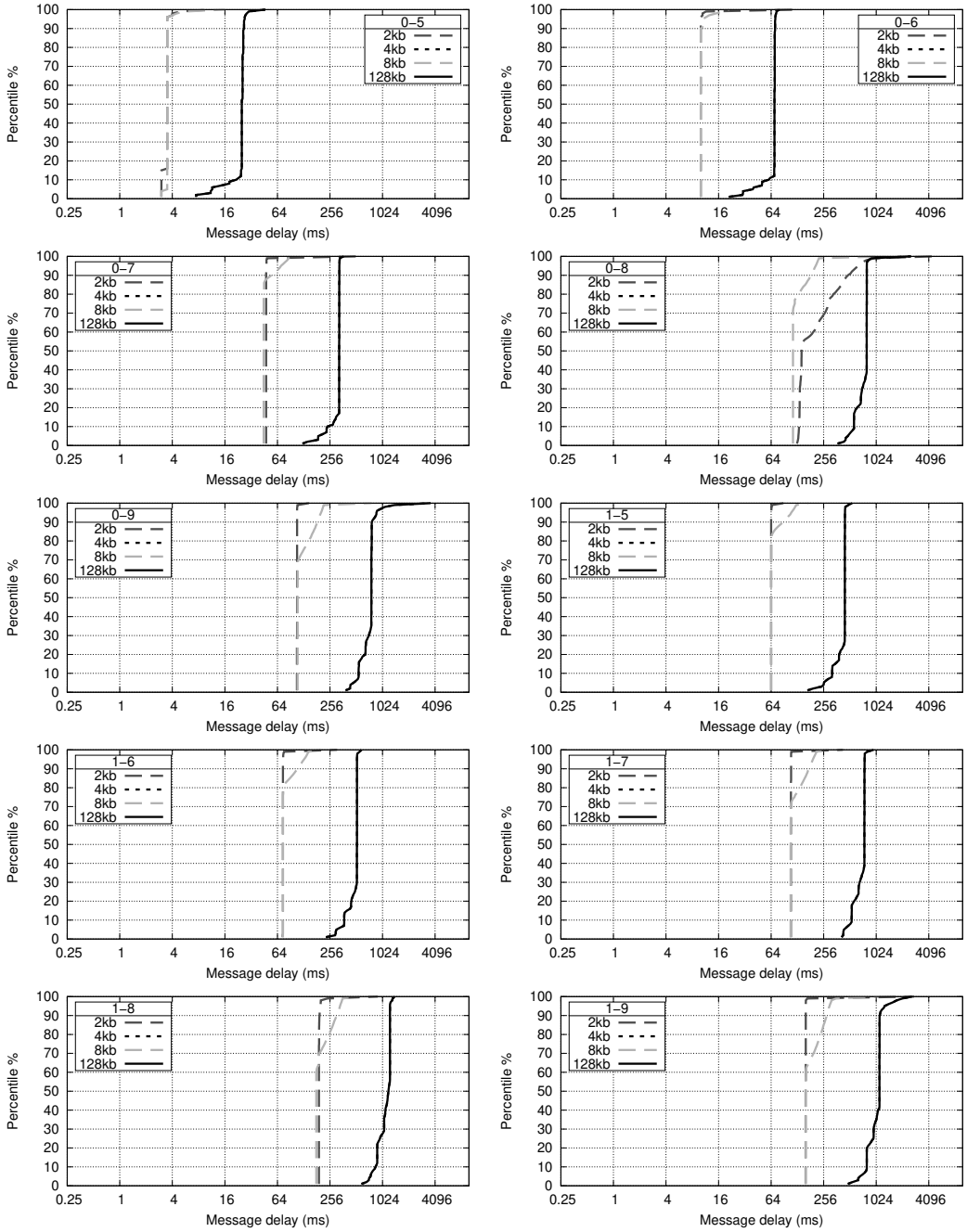

Figure 5.4. Communication delays with various message sizes between different AWS and DigitalOcean regions (x-axe is in log scale with the base 2).

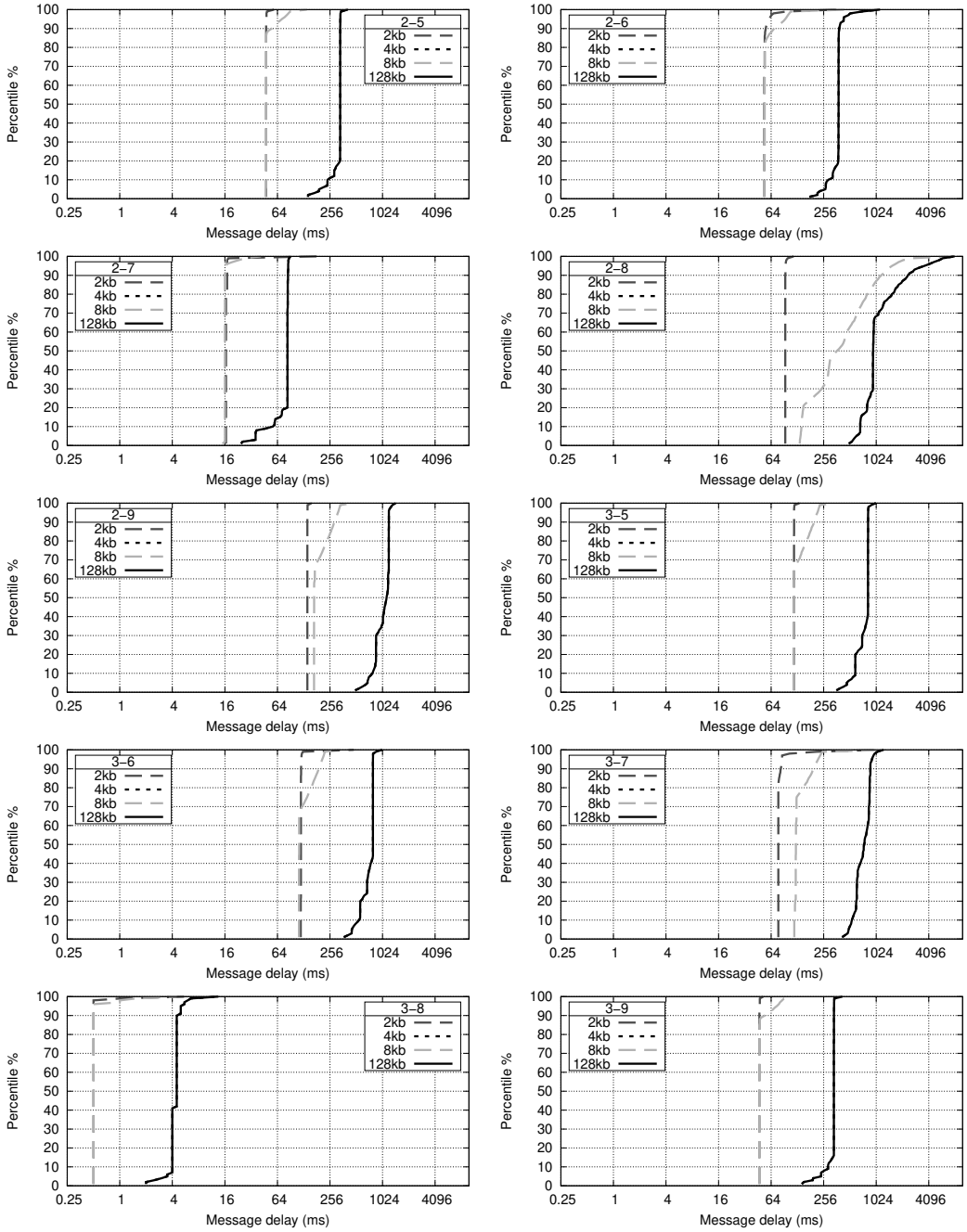

Figure 5.5. Communication delays with various message sizes between different AWS and DigitalOcean regions (x-axe is in log scale with the base 2).

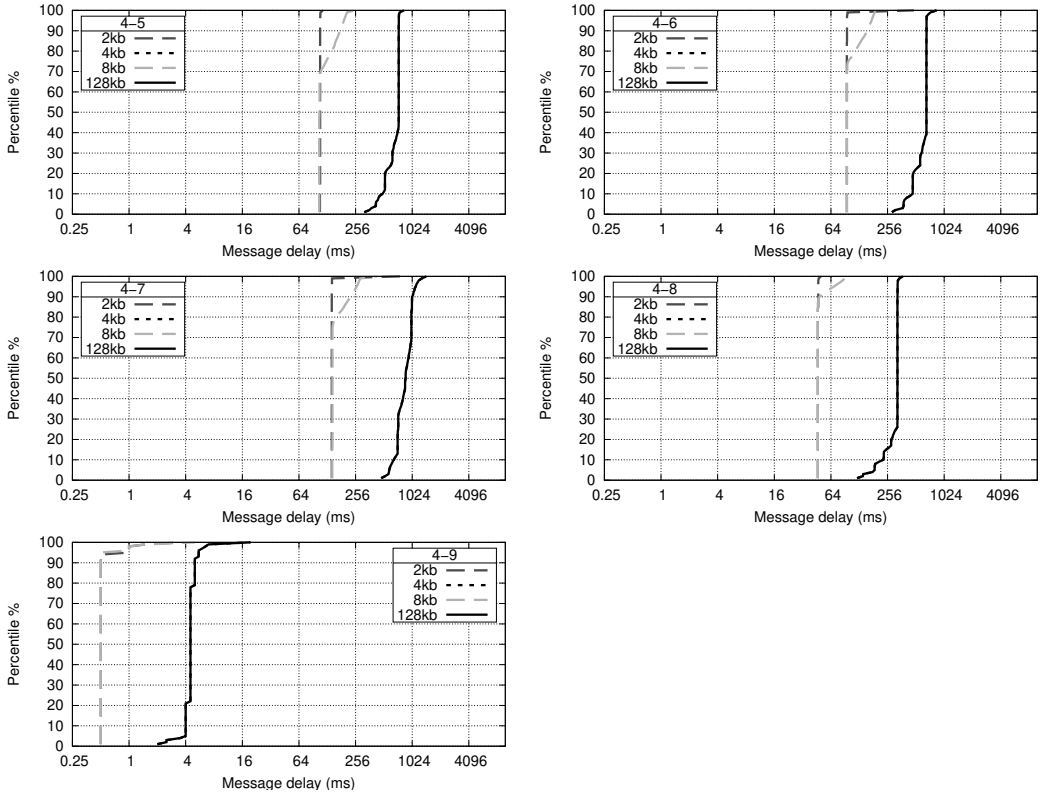

Figure 5.6. Communication delays with various message sizes between different AWS and DigitalOcean regions (x-axe is in log scale with the base 2).

5.2.2 The new model

This section defines the new timing model, the *hybrid synchronous* system model. This model sits between the synchronous and partially synchronous system models. Based on the experimental data presented in Section 5.2.1, we assume the synchronous system model for small messages and the partially synchronous system model for large messages.

Our new *hybrid synchronous* system model incorporates two communication properties, one for each message type:

- *Type $\mathscr{S}$ messages:* If an honest replica p sends a message m of type $\mathscr{S}$ to an honest replica q at time t, then q will receive m at time $t + \Delta_S$ or before.

- *Type $\mathscr{L}$ messages:* If an honest replica p sends a message m of type $\mathscr{L}$ to an honest replica q at time t, then q will receive m at time $max\{t, GST\} + \Delta_L$ or before.

Simply put, the model assumes small messages will always respect the specified time bound Δ_S, while large messages eventually respect the time bound Δ_L. Consequently, Δ_L does not need to be as conservative as Δ_S and can be set the same way as in partially synchronous protocols [24].

Lastly, we assume that all honest replicas start the execution within Δ_S time, and the communication between honest replicas is reliable: Every message an honest sender sends to an honest receiver is eventually received.

5.3 AlterBFT

In this section, we first describe the AlterBFT algorithm and its extensions (Sections 5.3.1–5.3.3). Then, we discuss the intuition behind its correctness (Section 5.3.4) and provide a formal proof (Section 5.3.5).

5.3.1 Protocol overview

AlterBFT provides a solution to the consensus problem under Byzantine failures (Section 2.2) in the hybrid synchronous model. It is similar to BoundBFT (Section 4.2.1) and the rotating leader version of Sync HotStuff [3; 5]. However, AlterBFT allows some messages (i.e., type $\mathscr{L}$ messages) to be arbitrarily delayed before GST. To enable this and still tolerate the same number of Byzantine failures ($f < n/2$) as synchronous algorithms, it must be designed with special care. Specifically, its agreement mechanism must solely depend on the timely delivery of type $\mathscr{S}$ messages, while the timely delivery of type $\mathscr{L}$ messages is needed just for progress.

Algorithms 7–9 present AlterBFT's variables and its pseudo-code for normal and abnormal case operations, respectively.

Algorithm 7 AlterBFT consensus algorithm: variables

```
1: Initialization:
2:     e_p := 0                                  ▷ the current epoch
3:     hasVoted := false                         ▷ has the replica voted in the current epoch?
4:     lockedVC_p := nil                         ▷ the most recent value certificate the replica is aware of
5:     epochsState_p[] := nil                    ▷ an epoch can be in one of the states: ACTIVE, COMMITTED, NOT-COMMITTED
6:     epochDecision_p[] := nil                  ▷ an epoch decision can be an id of a committed value or nil
```

AlterBFT's execution evolves as a sequence of epochs, numbered $0, 1, 2, ...$ Every epoch e has a designated leader, computed using a deterministic function. The leader is responsible for proposing a value in an epoch, and replicas vote for the proposed value. When a replica receives $f + 1$ votes signed by distinct replicas for a value v proposed by the leader in epoch e, the replica forms a value

Algorithm 8 AlterBFT consensus algorithm: normal case

```
1: when bootstrapping do StartEpoch(0)                                   ▷ the execution starts in epoch 0
2: Procedure StartEpoch(epoch) :                                         ▷ upon starting an epoch:
3:   e_p ← epoch; hasVoted_p ← false                                     ▷ the replica sets the current epoch, resets hasVoted variable, and...
4:   epochsState_p[e_p] ← ACTIVE; epochsDecision_p[e_p] ← nil;           ▷ sets epoch state and epoch decision to ACTIVE and nil, respectively
5:   if leader(e_p) = p then                                            ▷ if the replica is the leader in the current epoch, and...
6:     if e_p = 0 or lockedVC_p.epoch = e_p − 1 then                     ▷ e_p is the first epoch or the replica's lockedVC_p is from the previous epoch:
7:       Propose()                                                       ▷ the replica proposes immediately
8:     else                                                              ▷ otherwise,...
9:       execute Propose()   when timeoutEpochChange(e_p) expires        ▷ the replica waits for the timeout...
                                                                         ▷ to learn the most recent certified value, before proposing
10: Procedure Propose() :                                                ▷ in order to propose:
11:   v ← GetValue()                                                     ▷ the leader gets a value to propose, and...
12:   broadcast ⟨PROPOSE, e_p, v, lockedVC_p⟩                            ▷ broadcasts the proposal carrying the value and replica's lockedVC
13:   broadcast ⟨VOTE, e_p, id(v)⟩_p                                     ▷ then, it broadcasts a signed vote, and...
14:   hasVoted_p ← true                                                  ▷ sets hasVoted_p to avoid voting when it receives a proposal again
15: Function GetValue() :                                                ▷ GetValue function returns:
16:   if lockedVC_p = nil then                                           ▷ (a) if the leader does not know about any previously certified values,...
17:     return Value{payload : getPayload(), prev : nil}                 ▷ the new payload,
18:   if chainingEnabled() then                                          ▷ (b) in case the chaining is enabled,...
19:     return Value{payload : getPayload(), prev : lockedVC_p.id}       ▷ new payload linked to the most recent certified value, and
20:   return nil                                                         ▷ (c) nil, if there exist a certified value and chaining is disabled
21: when receive ⟨PROPOSE, e, v, VC⟩ and ⟨VOTE, e, id(v)⟩_c              ▷ when the replica receives a proposal and the vote for it...
22:  where e = e_p and c = leader(e) and epochsState_p[e_p] = ACTIVE do   ▷ signed by the leader of the current epoch that is active:
23:   if valid(v) ∧ v.prev = VC.id ∧ hasVoted_p = false ∧                ▷ if the value is valid, the replica hasn't voted in the current epoch, and...
24:    (condition_1 ∨ condition_2) then                                  ▷ one of the conditions is fullfilled:
25:     broadcast ⟨VOTE, e_p, id(v)⟩_p                                   ▷ the replica broadcast a VOTE message containing proposal id, and...
26:     hasVoted_p ← true                                                ▷ sets hasVoted_p so it does not vote twice
27:     forward ⟨VOTE, e, id(v)⟩_c                                       ▷ then, it (a) forwards the leader's vote, needed for timely equivocation detection, and...
28:     forward ⟨PROPOSE, e, v, VC⟩                                      ▷ (b) forwards the received proposal, needed for eventual delivery of all certified blocks
29: condition_1 ≡ (lockedVC_p = nil)                                     ▷ the replica is unaware of any certified value
30: condition_2 ≡ (lockedVC_p ≠ nil ∧ VC ≠ nil ∧ VC.epoch ≥ lockedVC_p.epoch) ▷ VC from proposal is more recent than lockedVC_p
31: when receive f + 1 distinct ⟨VOTE, e, id(v)⟩_* or ⟨QUIT-EPOCH, cert⟩ ▷ when the replica receives...
32:  where cert.type = VALUE-CERT                                        ▷ a value certificate:
33:   if ⟨QUIT-EPOCH, cert⟩ received then c ← cert                       ▷ it can receive it in a QUIT-EPOCH message, or...
34:   else c ← NewCert from f + 1⟨VOTE, e, id(v)⟩_*                      ▷ through f + 1 individual VOTE messages
35:   if c.epoch = e_p then                                              ▷ if the certificate is from the current epoch:
36:     lockedVC_p ← c                                                   ▷ the replica locks on it by updating its lockedVC_p
37:     if epochsState[e_p] = ACTIVE then                                ▷ then, if the replica has not received any other certificate yet...
38:       start timeoutCommit(e_p, c.id)                                 ▷ the replica starts timeoutCommit
39:     broadcast ⟨QUIT-EPOCH, c⟩                                        ▷ lastly, the replica broadcasts the certificate,...
40:     abort timeoutExtra(e_p)                                          ▷ aborts timeoutExtra if it was triggered, and...
41:     StartEpoch(e_p + 1)                                              ▷ starts the next epoch
42:   else                                                               ▷ in case the certificate is not from the current epoch:
43:     if leader(e_p) = p ∧                                             ▷ if the replica is current epoch leader, and...
44:      (lockedVC_p = nil ∨ c.epoch > lockedVC_p.epoch) then            ▷ the certificate is more recent than replica's lockedVC:
45:       lockedVC_p ← c                                                 ▷ the replica updates its lockedVC_p, and...
46:       broadcast ⟨QUIT-EPOCH, c⟩                                      ▷ broadcasts the new certificate
                                                                         ▷ ***First commit rule***
47: when timeoutCommit(e, id) expires do                                 ▷ when timeoutCommit expires:
48:   if epochsState[e] = ACTIVE then                                    ▷ if the replica did not observe any proof of misbehavior:
49:     epochsState[e] ← COMMITTED                                       ▷ the replica sets epoch state to COMMITTED, and...
50:     epochsDecision[e] ← id                                           ▷ the epoch decision value to id
                                                                         ▷ ***Second commit rule (FastAlterBFT)***
51: when receive ⟨VOTE, e, id(v)⟩_* from all replicas do                 ▷ when the replica receives votes from all replicas for the same value:
52:   if epochsState[e] = ACTIVE then                                    ▷ if the replica did not observe any proof of misbehavior:
53:     epochsState[e] ← COMMITTED                                       ▷ the replica sets epoch state to COMMITTED, and...
54:     epochsDecision[e] ← id(v)                                        ▷ the epoch decision value to id(v)
55: when receive ⟨PROPOSE, e, v, ∗⟩                                      ▷ when the replica receives a proposal...
56:  where epochDecision[e] = id(v) do                                   ▷ for a value corresponding to epoch's committed value:
57:   CommitValue(v)                                                     ▷ the replica commits value v
```

certificate $C_e(v)$. Every replica maintains the most recent value certificate it is aware of in variable $lockedVC$. Certificate $C_e(v)$ is more recent than $C_{e'}(v')$ if $e > e'$.

An honest leader l broadcasts the proposal with the most recent certified value it knows of, $lockedVC_l$, or a new value if $lockedVC_l = nil$ (line 12 in Algorithm 8). The proposal message is classified as a type $\mathscr{L}$ message because it carries a value that can be of arbitrary size. In addition to the proposal, the leader also broadcasts its signed vote for the proposal (line 13 in Algorithm 8). The vote contains the current epoch number and the hash of the value voted for, $id(v)$. Consequently, it is of a constant small size and is considered a type $\mathscr{S}$ message.

Upon receiving a proposal and a signed vote from the leader, a replica checks whether the proposal is valid (see Section 2.3) and votes for it if the proposed value truely extends the value from the leader's value certificate, and value certificate is at least as recent as the certificate in the replica's $lockedVC$ (lines 21–28 in Algorithm 8). The replica votes for a proposal by sending a signed vote message to all replicas and it votes only once per epoch. In addition, the replica forwards the proposal and the leader's vote to all replicas. Forwarding the leader's vote is needed to detect leader misbehavior, when a faulty leader sends different proposals to different replicas; forwarding the proposal ensures the eventual delivery of all certified values. If a value is certified, at least one replica among those who voted for the value is honest and will forward the value.

Upon receiving $C_e(v)$ for the current epoch e (lines 31–46 in Algorithm 8), an honest replica locks on it by setting $lockedVC$ to $C_e(v)$. After that, the replica propagates the certificate to all replicas, starts epoch $e+1$, and starts a timer, $timeoutCommit(e)$. The replica does this even if it has not yet received a proposal with value v since it knows it will receive it eventually (see Section 5.3.4).

When $timeoutCommit(e)$ expires and the replica did not receive any evidence of misbehavior (i.e., a different certificate, $C_e(\text{EQUIV})$ or $C_e(\text{BLAME})$), the replica sets $epochState[e]$ and $epochDecision[e]$ to COMMITTED and $id(v)$, respectively (lines 47–50 in Algorithm 8). Moreover, replica commits a value v, corresponding to $epochDecision[e]$, as soon as it receives it (lines 55–57 in Algorithm 8).

A replica r starts a $timeoutCertificate(e)$ timer when it enters the epoch to detect a malicious leader (line 2 in Algorithm 9). If the timer expires and r has not received any certificate yet, r broadcasts a message $\langle \text{BLAME}, e \rangle_r$, a type $\mathscr{S}$ message (lines 3–5 in Algorithm 9). When a replica receives $f+1$ blame messages for epoch e from distinct replicas (lines 6–9 in Algorithm 9), we say that the replica forms a blame certificate $C_e(\text{BLAME})$. Additionally, when an honest replica

receives votes for two distinct values from the leader in the same epoch e (lines 11–14 in Algorithm 9), the replica forms an equivocation certificate $C_e(\text{EQUIV})$.

Upon detecting a misbehavior (i.e., $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$) before committing a value in an epoch, the replica sets the epoch state to NOT-COMMITTED, broadcasts the certificate to all replicas (lines 16–20 in Algorithm 9), and moves to the next epoch $e+1$ (line 24 in Algorithm 9). Certificates are broadcast inside a $\langle$QUIT-EPOCH$\rangle$ messages. These messages are also considered as type $\mathscr{S}$ messages since certificates consist of multiple blame or vote messages that are of constant small size.

Algorithm 9 AlterBFT consensus algorithm: handling malicious leaders and asynchrony

```
1:  upon starting the epoch e do                                          ▷ when a replica enters a new epoch:
2:      start timeoutCertificate(e_p)                                     ▷ it starts the timer used to detect asynchrony or a malicious leader
3:  when timeoutCertificate(e) expires do                                 ▷ when timeoutCertificate expires:
4:      if e = e_p ∧ epochsState[e] = ACTIVE then                         ▷ if the replica did not receive any certificate:
5:          broadcast ⟨BLAME, e_p⟩_p                                      ▷ the replica broadcasts a BLAME message
6:  when receive f + 1 distinct ⟨BLAME, e⟩_* or ⟨QUIT-EPOCH, cert⟩        ▷ when a replica receives...
7:      where cert.type = BLAME-CERT do                                   ▷ a blame certificate:
8:          if ⟨QUIT-EPOCH, cert⟩ received then c ← cert                  ▷ it can receive it in a QUIT-EPOCH message with the certificate, or...
9:          else c ← NewCert from f + 1⟨BLAME, e⟩_*                       ▷ from f + 1 distinct BLAME messages
10:             MissbehaviorDetected(c, e)                                ▷ the replica calls MissbehaviorDetected with the certificate and epoch as parameters
11: when receive ⟨VOTE, e, id(v)⟩_c and ⟨VOTE, e, id(v')⟩_c or ⟨QUIT-EPOCH, cert⟩   ▷ when replica receives...
12:     where c = leader(e) and v ≠ v' or cert.type = EQUIV-CERT do        ▷ an equivocation certificate:
13:         if ⟨QUIT-EPOCH, cert⟩ received then c ← cert                  ▷ it can receive a QUIT-EPOCH message with the certificate, or...
14:         else c ← NewCert from ⟨VOTE, e, id(v)⟩_c and ⟨VOTE, e, id(v')⟩_c   ▷ two distinct VOTE messages signed by the epoch leader
15:         MissbehaviorDetected(c, e)                                    ▷ the replica calls MissbehaviorDetected with the certificate and epoch as parameters
16: Procedure MissbehaviorDetected(cert, e) :                             ▷ when MissbehaviorDetected is called in an epoch:
17:     if epochsState[e] = ACTIVE then                                   ▷ if the epoch is still active:
18:         epochsState[e] ← NOT-COMMITTED                                ▷ the replica sets state to NOT-COMMITTED
19:         if e = e_p then                                               ▷ moreover, if cert is the first certificate for the current epoch:
20:             broadcast ⟨QUIT-EPOCH, cert⟩                              ▷ the replica broadcasts the certificate, and...
21:             start timeoutExtra(e_p)                                   ▷ triggers timeoutExtra(e_p)
22: when timeoutExtra(e) expires do                                       ▷ when timeoutExtra expires:
23:     if e = e_p then                                                   ▷ if the replica is in epoch e:
24:         StartEpoch(e_p + 1)                                           ▷ the replica starts the next epoch
```

5.3.2 FastAlterBFT

We now extend AlterBFT with an optimistic fast path [4; 10; 52; 68]. Namely, an honest replica commits a value v if it receives votes for v in epoch e from all replicas in the system before detecting any evidence of misbehavior (i.e., $C_e(\text{EQUIV})$ or $C_e(\text{BLAME})$). Consequently, in optimistic conditions, when there are no failures in the system, AlterBFT commits a value in only two communication steps, without waiting for any synchrony bound Δ. Optimistic fast paths have historically introduced significant complexity and potential bugs in partially synchronous

protocols [1; 2]. AlterBFT's optimistic fast path, however, required changes in a few lines of the non-optimistic protocol. Namely, a simple commit rule (lines 51-54 in Algorithm 8) and an additional timeout, $timeoutExtra(e)$. This timeout is triggered in case the first certificate received in an epoch is $C_e(\text{EQUIV})$ or $C_e(\text{BLAME})$ (line 21 in Algorithm 9). The replica then moves to the next epoch either when this timeout expires (lines 22-24 in Algorithm 9) or if it receives a value certificate in the meantime (lines 40-41 in Algorithm 8).

5.3.3 AlterBFT as a blockchain protocol

We extended AlterBFT to build a blockchain (Section 2.3) inspired by the idea introduced in [119]. Namely, in each epoch the leader will not propose the most recent certified block (value), B_k, it is aware of. Instead, it will propose a new block B_{k+1} that extends B_k (lines 18–19 in Algorithm 8). Replicas will accept B_{k+1} if it truly extends B_k and if $C_e(B_k)$ is more recent than their $lockedVC$ (line 30 in Algorithm 8). Since blocks are linked, committing block B_k commits all blocks it extends. Specifically, when a replica commits block B_k proposed in the same epoch, we say it directly commits block B_k and indirectly commits all B_k's ancestors. Moreover, the protocol ensures that whenever an honest replica commits block B_k in epoch e, only blocks extending B_k will be certified and committed in the following epochs. This way, the consistency of the blockchain is maintained and forks are prevented. Lastly, since AlterBFT is a non-stopping protocol, new blocks will be added, and the blockchain creation will continue.

5.3.4 Correctness intuition

In this section, we provide the intuition on how AlterBFT maintains protocol correctness in the hybrid synchronous model.

Epoch synchronization

The epoch synchronization mechanism in AlterBFT is similar to that in BoundBFT (Section 4.2.2). Namely, honest replicas use $timeoutCertificate(e)$ (line 2 in Algorithm 9) to unfailingly form one of the certificates in each epoch regardless of Byzantine behavior. And then forward the certificates to synchronize replicas (lines 39 in Algorithm 8 and 20 in Algorithm 9).

In BoundBFT, the equivocation certificate contains equivocated proposals. Additionally, an honest replica must receive both a proposal and $f + 1$ votes for the replica to form a value certificate. Since proposals carry values that can

be of variable size, messages carrying these certificates must be categorized as type $\mathscr{L}$ messages. As a result, in the hybrid synchronous model, BoundBFT's epoch synchronization would not work since before GST type $\mathscr{L}$ messages can be arbitrarily delayed.

In AlterBFT, however, we modified the certificates to exclude the actual values. Specifically, the equivocation certificate now consists of two signed leader vote messages instead of two proposal messages (line 11 in Algorithm 9). Additionally, a replica forms a value certificate when it receives $f + 1$ votes for a proposal, regardless of whether it has received the proposal itself (line 31 in Algorithm 8). The former change is sufficient for equivocation detection, as only the leader can sign its vote message, and honest replicas accept the proposal only after receiving a signed vote from the leader (line 21 in Algorithm 8). In the latter case, a replica does not need to wait for the proposal after receiving $f + 1$ votes because it knows that at least one of the replicas, among those $f + 1$ that voted for the proposal, is honest and will forward the certificate (line 28 in Algorithm 8).

As a result, messages with certificates are considered as type $\mathscr{S}$ and are delivered within Δ_S time. Allowing replicas to be synchronized and to access each epoch within Δ_S time.

Agreement

AlterBFT ensures that no two honest replicas commit different values. AlterBFT's agreement relies on two invariants: if an honest replica r commits value v in epoch e then (i) $C_e(v)$ is the only value certificate that exists in epoch e (i.e., no honest replica voted for a value $v' \neq v$ in e), and (ii) all honest replicas locked on v by setting $lockedVC$ to $C_e(v)$ in the epoch e. As a result, honest replicas only vote for values certified in epochs $e' \geq e$ in all following epochs. Since by (i) v is the only certified value in e and by (ii) all honest set $lockedVC$ to $C_e(v)$, in epochs greater than e honest replicas will only vote for v and no other value $v' \neq v$ will be certified and committed.

Replica r commits value v if (1) $timeoutCommit(e)$ expires (line 50 in Algorithm 8) or (2) r receives votes from all replicas (line 54 in Algorithm 8), and no misbehavior is detected. In (1), invariant (i) holds because r, upon receiving $C_e(v)$ at time t, starts $timeoutCommit(e)$ and broadcasts $C_e(v)$ (lines 38–39 in Algorithm 8). Since a message with $C_e(v)$ is a type $\mathscr{S}$ message, in Δ_S time, all honest replicas will receive it. If any honest replica q voted for $v' \neq v$, it must have done so before $t + \Delta_S$. Since q also forwards the leader's vote for v' (line 27 in Algorithm 8), r receives it before $t + 2\Delta_S$. As a result, an honest replica

must set $timeoutCommit(e)$ to expire in $2\Delta_S$. This way, replica r receives the leader's votes for v and v' before $timeoutCommit(e)$ expires. Consequently, it forms an equivocation certificate $C_e(\text{EQUIV})$ (line 11 in Algorithm 9) and does not commit. Similarly, invariant (ii) holds because q will only lock on $C_e(v)$ if q moved to epoch $e+1$ upon receiving $C_e(\text{EQUIV})$ or $C_e(\text{BLAME})$ before $t+\Delta_S$ (line 24 in Algorithm 9). Since q will forward the received certificate (line 20 in Algorithm 9) and messages carrying certificates are type $\mathscr{S}$, r will receive it before $timeoutCommit(e) = 2\Delta_S$ expires and will not commit (line 18 in Algorithm 9).

Invariant (i) trivially holds in (2) because a replica knows that all honest replicas voted for v since it received votes from all honest replicas, and each honest replica votes only once. However, achieving (ii) needs extra care. In this case, a replica r locks on value v at time t and commits at time t', where $t < t' < t + timeoutCommit(e)$. As a result, if some honest replica q receives $C_e(\text{EQUIV})$ or $C_e(\text{BLAME})$ at t'', $t' - \Delta_S \leq t'' < t' + \Delta_S$, it will move to epoch $e+1$ without locking on $C_e(v)$ and replica r will not be aware of this. To handle this scenario, an honest replica q will set $timeoutExtra(e)$ to expire in $2\Delta_S$ after receiving a $C_e(\text{EQUIV})$ or $C_e(\text{BLAME})$ (line 21 in Algorithm 9), and it will move to the next epoch only if it receives $C_e(v)$ or if the $timeoutExtra(e)$ expires (lines 41 in Algorithm 8 and 24 in Algorithm 9). Since the replica q's $timeoutExtra(e)$ expires at $t'' + 2\Delta_S$ and $t'' + 2\Delta_S > t + \Delta_S$ q will receive $C_e(v)$ and lock on it before moving to the next epoch, which ensures invariant (ii).

Progress

AlterBFT ensures that all correct replicas eventually commit a value. AlterBFT guarantees progress after GST (Section 5.2.2) in the first epoch of an honest leader. Namely, the progress is guaranteed in epoch $e > GST$ with an honest leader if: (1) the leader proposes a value that all honest replicas vote for, and (2) no honest replica broadcasts a BLAME message in epoch e. While (1) ensures the value certificate is formed and the $timeoutCommit(e)$ started, (2) ensures that no blame certificate is possible. In addition, since the honest leader proposes only one value, no $C_e(\text{EQUIV})$ is possible. As a result, when $timeoutCommit(e)$ expires, all honest replicas will commit the proposed value.

To ensure (1), the honest leader must learn the most recent certified value before proposing. Consequently, upon entering epoch e at time t, if it does not have a value certificate from the previous epoch, $e-1$, the honest leader l starts $timeoutEpochChange(e)$ (line 9 in Algorithm 8). Since honest replicas enter epoch e by time $t+\Delta_S$, they all broadcast their $lockedVC$ (line 39 in Algorithm 8)

at $t+\Delta_S$ at the latest. As a result, the leader l will receive these certificates by the time $t+2\Delta_S$. Consequently, the honest leader must set $timeoutEpochChange(e)$ to $2\Delta_S$ to make sure it will learn the most recent certified value.

To guarantee (2), honest replicas must receive the value certificate before $timeoutCertificate(e)$ expires. If an honest replica r starts epoch e at time t, the honest leader l will start the epoch e at $t+\Delta_S$ at the latest. The leader might wait for the $timeoutEpochChange(e) = 2\Delta_S$ before proposing, and as a result, it will propose a value by time $t+\Delta_S+2\Delta_S$. Since the proposal is a message of type $\mathscr{L}$, it may require Δ_L time to reach all honest replicas. Consequently, all honest replicas will vote for the value by time $t+3\Delta_S+\Delta_L$. Finally, since the votes are type $\mathscr{S}$, the $timeoutCertificate(e)$ should account for an additional Δ_S. In summary, to guarantee (2) honest replicas must set their $timeoutCertificate(e)$ to $4\Delta_S+\Delta_L$.

5.3.5 Correctness proof

This section presents the proof of AlterBFT. While this proof closely mirrors that of BoundBFT (Section 4.2.3), maintaining a similar structure with corresponding theorems and lemmas, it is provided here in its entirety for the sake of completeness.

Lemma 7. *Every honest replica always progresses to the next epoch.*

Proof. Assume, for the sake of contradiction, that there exists an honest replica r that remains in some epoch e indefinitely. This would imply that in epoch e, r did not generate any of the certificates $C_e(B_k)$, $C_e(\text{BLAME})$, or $C_e(\text{EQUIV})$.

However, upon entering epoch e, every honest replica starts the $timeoutCertificate(e)$ timer (line 2 in Algorithm 9). When this timeout expires, if an honest replica has not received any certificate, it broadcasts the BLAME message (lines 3–5 in Algorithm 9).

Therefore, if no certificate is formed before the $timeoutCertificate(e)$ expires, all honest replicas will broadcast the BLAME message, resulting in the formation of the blame certificate $C_e(\text{BLAME})$. This contradicts the assumption that an honest replica can stay in epoch e indefinitely. Thus, every honest replica must progress to the next epoch. □

Lemma 8. *If an honest replica starts epoch e at time t, then all honest replicas will start epoch e by time $t+\Delta_S$.*

Proof. Suppose an honest replica r starts epoch e at time t. This implies that r either received and broadcast $C_{e-1}(B_k)$ at time t (line 39 in Algorithm 8), or re-

ceived and broadcast C_{e-1}(BLAME) or C_{e-1}(EQUIV) at time $t - timeoutExtra(2\Delta_S)$ (line 20 in Algorithm 9).

Since messages with certificates (QUIT-EPOCH messages) are of type $\mathscr{S}$, they will be delivered within Δ_S time. Therefore, in the first case, all honest replicas receive $C_{e-1}(B_k)$ by time $t + \Delta_S$ and start epoch e. In the second case, all honest replicas receive C_{e-1}(BLAME) or C_{e-1}(EQUIV) by time $t - \Delta_S$ and subsequently start epoch e within $2\Delta_S$, resulting in the same deadline of $t + \Delta_S$.

It is also possible that while an honest replica is waiting for $timeoutExtra(e-1)$ to expire, it may receive $C_{e-1}(B_k)$. In such a case, the replica will abort the timeout, broadcast $C_{e-1}(B_k)$, and start epoch e (lines 39–41 in Algorithm 8). All honest replicas will then receive this message and, if they have not already done so, will start epoch e.

Therefore, all honest replicas start epoch e by time $t + \Delta_S$. □

Theorem 5. *(Epoch synchronization) All honest replicas continuously move through epochs, with each replica starting a new epoch within Δ_S time of any other honest replica.*

Proof. We prove this theorem by combining Lemma 7 and Lemma 8.

First, from Lemma 7, we know that every honest replica always moves to the next epoch. This ensures that no honest replica remains stuck in any epoch indefinitely.

Second, from Lemma 8, we know that if an honest replica starts epoch e at time t, then all honest replicas start epoch e by time $t + \Delta_S$. This guarantees that all honest replicas start each epoch within Δ_S time of each other.

Combining these two results, we can conclude that all honest replicas continuously move through epochs, with each replica initiating a new epoch within Δ_S time of any other honest replica. □

Lemma 9. *If an honest replica directly commits block B_k in epoch e, then (i) no block different from B_k can be certified in epoch e, and (ii) every honest replica locks on block B_k in epoch e.*

Proof. AlterBFT has two commit rules. We need to show that the lemma holds in both scenarios.

First, consider the general case where an honest replica r directly commits B_k at time t because $timeoutCommit(e) = 2\Delta_S$ expired and it did not receive any blame or equivocation certificate (lines 47–50 in Algorithm 8). This implies that at time $t - 2\Delta_S$, r received $C_e(B_k)$, locked on it, and started $timeoutCommit(e)$. Additionally, r broadcast $C_e(B_k)$. Since this message is of type $\mathscr{S}$, all honest replicas received $C_e(B_k)$ within Δ_S time, by $t - \Delta_S$.

For part (i), assume for contradiction that an honest replica q received and voted for a block $B_l \neq B_k$ in epoch e. Since every honest replica votes only once, q must have received the proposal and leader's vote for B_l before receiving $C_e(B_k)$, i.e., at time $t_1 < t - \Delta_S$. Upon voting for B_l, q broadcast the leader's vote (line 27 in Algorithm 8). Consequently, r would receive the leader's vote for B_l by $t_1 + \Delta_S$, which is before t. Moreover, since r received $C_e(B_k)$ at $t - 2\Delta_S$, we know that at least one honest replica voted for B_k at some moment $t_2 < t - 2\Delta_S$. Therefore, r would receive the leader's vote for B_k by $t_2 + \Delta_S$. Since both leader's votes for B_k and B_l would arrive at r before t, a $C_e(\text{EQUIV})$ certificate would be constructed, and r would not commit (line 18 in Algorithm 9). This is a contradiction. Therefore, property (i) holds as no honest replica votes for a block different from B_k, otherwise r would not commit.

For part (ii), it suffices to prove that every honest replica receives $C_e(B_k)$ before moving to the next epoch. This is sufficient because, due to (i), B_k is the only certified block in epoch e, and since e is the current epoch, there is no more recent block certificate. Consequently, if an honest replica receives $C_e(B_k)$ in epoch e, it will update its $lockedVC$ to it (line 36 in Algorithm 8). Since we know all honest replicas will receive $C_e(B_k)$ by $t - \Delta_S$, we need to prove that no honest replica will start epoch $e + 1$ before $t - \Delta_S$.

Assume, for contradiction, that an honest replica q moves to epoch $e + 1$ at $t_1 < t - \Delta_S$ without receiving $C_e(B_k)$. Since $C_e(B_k)$ is the only block certificate in epoch e, q must have moved to epoch $e + 1$ because it received $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$. Since q broadcasts $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ (line 20 in Algorithm 9) at time t_1, r would receive them by $t_1 + \Delta_S$. Since $t > t_1 + \Delta_S$, r would not commit B_k, a contradiction. Note that waiting for $timeoutExtra(e) = 2\Delta_S$ (line 21 in Algorithm 9) after receiving $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ is not needed in this case.

Now consider the case where r commits due to the FastAlterBFT commit rule (lines 51–54 in Algorithm 8). Specifically, this means r starts $timeoutCommit(e)$ at $t - 2\Delta_S$ and commits at some moment $t_1 < t$ after receiving votes from all replicas. Part (i) trivially holds because if r received votes for B_k from all replicas, this means that all honest replicas (f+1) voted for B_k. Since honest replicas vote only once in an epoch, no other $B'_k \neq B_k$ can collect (f+1) votes and be certified in epoch e.

For part (ii), every replica needs to wait $timeoutExtra(e) = 2\Delta_S$ before moving to the next epoch in case it receives $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$ first (line 21 in Algorithm 9). Assume, for contradiction, that an honest replica q moved to epoch $e + 1$ before receiving $C_e(B_k)$, namely before $t - \Delta_S$. Again, due to (i), replica q moved to epoch $e + 1$ because it received $C_e(\text{BLAME})$ or $C_e(\text{EQUIV})$. Due

to the extra $2\Delta_S$ timeout, it must have received one of these certificates at some moment $t_2 < t - \Delta_S - 2\Delta_S$. Since q would forward the received certificate at t_2, all honest replicas, including r, would receive this certificate by $t_2 + \Delta_S$, and since this is before $t - 2\Delta_S$, r would not start $timeoutCommit(e)$ at $t - 2\Delta_S$ and would not commit, a contradiction.

Therefore, both parts (i) and (ii) hold. □

Lemma 10. *If $C_e(B_k)$ is the only certified block in epoch e and $f+1$ honest replicas lock on block B_k in epoch e, then in all epochs $e' > e$ these replicas will only vote for blocks that extend B_k.*

Proof. Let set C contain $f+1$ or more honest replicas that lock on B_k in epoch e. We prove this lemma by induction on the epoch number.

Base step ($e' = e+1$) : A replica $r \in C$ will only vote for a block $B_{k'}$ in epoch e' if $B_{k'}$ extends a block certified in an epoch greater than or equal to e (line 30 in Algorithm 8). Since e is the previous epoch and the highest in the system, and B_k is the only certified block in epoch e, the lemma holds trivially for $e' = e+1$.

Induction step ($e' \rightarrow e'+1$): Assume the lemma holds for until epoch $e'+1$. We will show it holds for $e'+1$ also.

From the induction hypothesis, in epochs $e+1$ to $e'+1$, replicas in C only vote for blocks that extend B_k. Let B_l be the last block to receive $f+1$ vote messages in some epoch e'' where $e+1 \leq e'' \leq e'-1$. Therefore, for all replicas in C, $lockedVC = C_{e''}(B_l)$ and it follows that B_l extends B_k. As a result, a replica will only vote for a block $B_{k'}$ in e' if $B_{k'}$ extends B_l and therefore B_k.

By induction, the lemma holds for all epochs $e' > e$. □

Lemma 11. *If an honest replica directly commits block B_k in epoch e, then any block B_l that is certified in epoch $e' > e$ must extend B_k.*

Proof. The proof follows directly from Lemmas 9 and 10. More precisely, if an honest replica directly commits block B_k in epoch e, by Lemma 9, we know that $f+1$ honest replicas (set C) lock on block B_k in epoch e and B_k is the only certified block in epoch e. Consequently, by Lemma 10, replicas from C vote only for the blocks extending block B_k in epochs $e' > e$. Therefore, no block B_l that does not extend B_k can collect $f+1$ votes and thus cannot be certified in any epoch $e' > e$. □

Theorem 6. *(Agreement) No two honest replicas commit different blocks at the same height.*

Proof. Suppose, for the sake of contradiction, that two distinct blocks B_k and B'_k are committed for the height k. Suppose B_k is committed as a result of B_l being

directly committed in epoch e and B'_k is committed as a result of $B_{l'}$ being directly committed in epoch e'. Without loss of generality, assume $l < l'$. Note that all directly committed blocks are certified. This is true because both commit rules require that replica receives $C_e(B_k)$ before directly commiting B_k in epoch e (lines 47 and 51 in Algorithm 8). By Lemma 11, $B_{l'}$ extends B_l. Therefore, $B_k = B'_k$ which is a required contradiction.

□

Lemma 12. *If the epoch e is after GST and the leader of the epoch is an honest replica, all honest replicas commit a block in this epoch.*

Proof. Consider an epoch e with an honest leader l, occurring after GST. Let $t > GST$ be the time when the first honest replica starts epoch e. By Lemma 8, all honest replicas enter epoch e by time $t + \Delta_S$. Consequently, they all broadcast their $lockedVC$ by time $t + \Delta_S$ at the latest. As a result, l will receive certificates from all honest replicas by time $t + 2\Delta_S$. This is why l needs to wait for $timeoutEpochChange(e) = 2\Delta_S$ after entering the epoch if it does not know the certificate from the previous epoch, to update its $lockedVC$ to the most recent certificate (lines 9 and 42–46 in Algorithm 8).

Consequently, the honest leader l broadcasts $\langle \text{PROPOSE}, e, B_k, lockedVC_l \rangle$ and $\langle \text{VOTE}, e, id(B_k) \rangle_l$ by time $t + 3\Delta_S$ at the latest. Since we are after GST, all honest replicas receive both messages within Δ_L time, by time $t + 3\Delta_S + \Delta_L$ and vote for the proposal. The votes are of type $\mathscr{S}$ and all honest replicas receive them within Δ_S time, form a block certificate, and start $timeoutCommit(e)$ by time $t + 4\Delta_S + \Delta_L$.

Given that the earliest point when an honest replica entered epoch e is t and honest replicas set $timeoutCertificate(e)$ to expire in $4\Delta_S + \Delta_L$, no honest replica will send a $\langle \text{BLAME}, e \rangle_*$ message in epoch e, and $C_e(\text{BLAME})$ cannot be formed. Furthermore, since l is honest, it does not equivocate, so no $C_e(\text{EQUIV})$ can be formed in epoch e either.

Consequently, when $timeoutCommit(e)$ expires, all honest replicas will commit B_k and all its ancestors. □

Theorem 7. *(Progress) All honest replicas keep committing new blocks.*

Proof. By the Theorem 5 replicas move through epochs. Eventually, after GST, replicas will reach epochs with honest leaders. Consequently, by the Lemma 12 all honest replicas will commit blocks in these epoch. □

Lemma 13. *Every block B_k (where $k \neq 0$) proposed by an honest replica in some epoch e has, as its ancestors, blocks that have been certified in one of the epochs $e' < e$.*

Proof. The proof for this lemma directly follows from Algorithm 8. Specifically, a leader l of epoch e that proposes block B_k, which extends some block B_l, must provide a valid block certificate for block B_l from some epoch $e' < e$ (line 10 in Algorithm 8).

Furthermore, an honest replica will only vote for B_k if $B_k.prev = id(B_l)$ (line 23 in Algorithm 8). □

Theorem 8. *(Block availability) All blocks committed by honest replicas will eventually be received by all honest replicas.*

Proof. Assume an honest replica r commits block B_k. We know that B_k must be certified before being committed (lines 38, 47, and 51 in Algorithm 8). By Lemma 13, all of B_k's ancestors are also certified blocks.

For a block to be certified, at least one honest replica must vote for it. Additionally, an honest replica, along with the vote, forwards the proposal (line 28 in Algorithm 8). Consequently, if a block is certified at time t, at least one honest replica forwards the proposal before time t.

Since the PROPOSE message is of type $\mathscr{L}$, we know, by the communication properties of type $\mathscr{L}$ messages (Section 5.2.2), that it will be received by all honest replicas before $max\{t, GST\} + \Delta_L$. □

Theorem 9. *(External validity) Every committed block satisfies the predefined* valid() *predicate.*

Proof. We know that block must be certified before being committed (lines 38, 47, and 51 in Algorithm 8). By Lemma 13, all of B_k's ancestors are also certified blocks. This implies that at least one honest replica accepted these blocks, meaning that $valid()$ returned true for these blocks on at least one honest replica (line 23 in Algorithm 8). □

5.4 Experimental evaluation

In this section, we first introduce the experimental setup (Section 5.4.1), and discuss message sizes in AlterBFT (Section 5.4.2). Then we evaluate the performance (throughput and latency) in fault-free scenarios (Section 5.4.3) as well as under faults (Section 5.4.4). Lastly, we evaluate hybrid model alternatives (Section 5.4.5) and comment on the main takeaways (Section 5.4.6).

5.4.1 Evaluation setup

We used the same methodology as in Section 4.4.1. Namely, we implemented AlterBFT in Go and compared it to the same protocols as BoundBFT (see Table 4.2). Table 5.3 compares the good-case latency of the considered protocols in failure-free executions.

	Good-case latency
Sync HotStuff [5]	$\delta_L+\delta_S+2\Delta$
Tendermint [20]	$\delta_L+2\delta_S$
HotStuff-2 [82]	$3\delta_L+2\delta_S$
AlterBFT (this paper)	$\delta_L+\delta_S+2\Delta_S$
FastAlterBFT (this paper)	$\delta_L+\delta_S$

Table 5.3. Protocols in our evaluation and their good-case latencies. δ_L is the actual delay of large messages (i.e., blocks); δ_S is the actual delay of small messages (i.e., votes and certificates); Δ is the conservative message delay that accounts for large and small messages; and Δ_S is the conservative delay of small messages.

We conducted our experiments on Amazon EC2 and evenly distributed replicas across 5 AWS regions: North Virginia (us-east-1), São Paulo (sa-east-1), Stockholm (eu-north-1), Singapore (ap-southeast-1), and Sydney (ap-southeast-2). Replicas were hosted on *t3.medium* instances, with 2 virtual CPUs, 4GB of RAM, and running Amazon Linux 2.

5.4.2 On message size

The hybrid synchronous model in AlterBFT differentiates between two types of messages: $\mathscr{S}$ and $\mathscr{L}$. Based on our experimental evaluation (Section 3.3.1), messages of 4 KB or smaller are classified as type $\mathscr{S}$, while larger messages are classified as type $\mathscr{L}$.

Table 5.4 presents the sizes of all messages exchanged in AlterBFT. The VOTE and BLAME messages are small, fixed-size messages that belong to type $\mathscr{S}$. In contrast, the PROPOSE and QUIT-EPOCH messages have variable sizes.

QUIT-EPOCH messages, which carry certificates, must be exchanged in a timely manner for correctness (Section 5.3.5) and are thus classified as type $\mathscr{S}$. The size of a certificate depends on a majority quorum of replicas. In our experiments,

Message	Message size (payload)	Message type
PROPOSE	variable size, dominated by block size	$\mathscr{L}$
VOTE	fixed size, below 120 bytes	$\mathscr{S}$
BLAME	fixed size, below 100 bytes	$\mathscr{S}$
QUIT-EPOCH	fixed size (50 bytes) + quorum-size * 66 bytes	$\mathscr{S}$

Table 5.4. Message sizes in the AlterBFT prototype.

the largest certificate in a system with 85 replicas is 2.8 KB. Consequently, with the current prototype, we can accommodate deployments with up to 120 replicas. For larger systems, a more optimized signature techniques such as BLS [19] would be necessary.

The size of PROPOSE messages depends on both the block and certificate sizes. However, in AlterBFT, these messages are classified as type $\mathscr{L}$, which means there are no restrictions on their size, and consequently, no restrictions on the block size as well.

5.4.3 Failure-free performance

In this section, we compare AlterBFT to state-of-the-art protocols in the absence of failures. We measure latency and throughput while varying the system size (i.e., 25, 55, and 85 replicas) and block size (i.e., from 1 KB up to 1 MB).

Latency

Figure 5.7 shows the average consensus latency computed by leaders. From Table 5.3, Sync HotStuff and AlterBFT latencies directly depend on conservative synchronous bounds. This is because both protocols wait for a timeout (i.e., $timeoutCommit$, computed as twice the time bound) before committing a value. We used the conservative 99.99% [80] values as synchronous bounds for Sync HotStuff and AlterBFT. Table 5.5 shows 99.99% delays for messages of different sizes. We can clearly see that values increase with the message size, from 254 ms to 6099 ms for 1 KB and 1 MB message sizes, respectively.

Message size (KB)	1	2	3	4	8	16	32	64	128	256	512	1024
99.99 % (ms)	254	273	308	325	514	663	995	2594	2825	3935	5080	6099

Table 5.5. The 99.99 % percentile of collected message delays for different message sizes; values collected during one-day-long experiments.

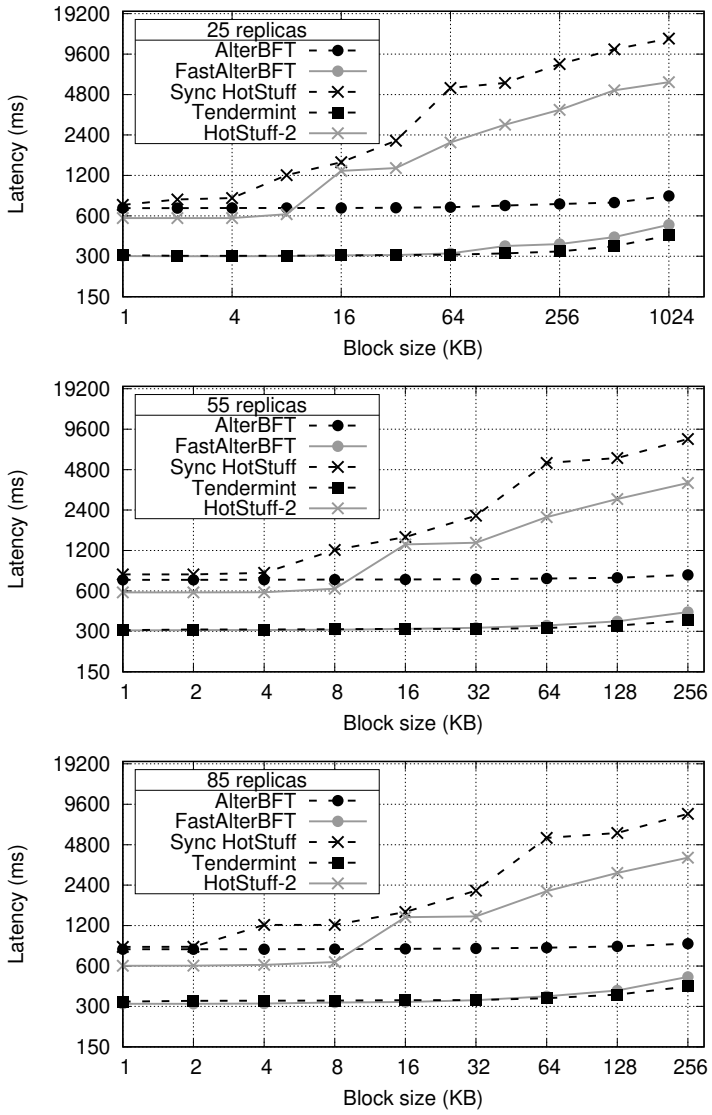

Figure 5.7. Average latency for all protocols when varying system size (i.e., 25, 55, and 85 replicas) and block size (all graphs in log scale).

Classical synchronous protocols, such as Sync HotStuff, must adopt a Δ that accommodates the timely delivery of all messages, regardless of their size. As a result, the Δ must be sufficiently large to ensure the delivery of both large and small messages. Consequently, because the size of messages carrying blocks increases with the block size, the Δ also increases (see Table 5.6), leading to higher latency in Sync HotStuff.

In contrast, AlterBFT's $timeoutCommit$ consists of Δ_S, which accounts only

Block size (KB)	1	2	4	8	16	32	64	128	256	512	1024
25 replicas	273 ms (2 KB)	308 ms (3 KB)	325 ms (5 KB)	514 ms (9 KB)	663 ms (17 KB)	995 ms (33 KB)	2594 ms (65 KB)	2825 ms (129 KB)	3935 ms (257 KB)	5080 ms (513 KB)	6099 ms (1025 KB)
55 replicas	308 ms (3 KB)	325 ms (4 KB)	325 ms (6 KB)	514 ms (10 KB)	663 ms (18 KB)	995 ms (34 KB)	2594 ms (66 KB)	2825 ms (130 KB)	3935 ms (258 KB)	5080 ms (514 KB)	6099 ms (1026 KB)
85 replicas	325 ms (4 KB)	325 ms (5 KB)	514 ms (7 KB)	514 ms (11 KB)	663 ms (19 KB)	995 ms (35 KB)	2594 ms (67 KB)	2825 ms (131 KB)	3935 ms (259 KB)	5080 ms (515 KB)	6099 ms (1027 KB)

Table 5.6. Sync HotStuff's synchronous conservative bound Δ for different block and system sizes. Table's fields show: Δ (message size it accounts for).

for the timely delivery of small type $\mathscr{S}$ messages. Since messages carrying blocks are categorized as type $\mathscr{L}$ messages, Δ_S remains unaffected by block size (see Table 5.7). Consequently, the difference between AlterBFT's and Sync HotStuff's latencies increases with the block size. Specifically, up to 4 KB block size AlterBFT performs slightly better, however already with 8 KB blocks AlterBFT's latency is more than 1.5× lower than Sync HotStuff's, and this difference raises to an outstanding 14.9× with 1 MB blocks.

Block size (KB)	1	2	4	8	16	32	64	128	256	512	1024
25 replicas	254 ms (1 KB)										
55 replicas	273 ms (2 KB)										
85 replicas	308 ms (3 KB)										

Table 5.7. AlterBFT's synchronous conservative bound Δ_S for different block and system sizes. Table's fields show: Δ_S (message size it accounts for).

On the other side, Figure 5.7 shows that Tendermint's latency is still lower, around 2×, than AlterBFT's in all setups. However, the difference does not increase with the block size. This is because Tendermint's and AlterBFT's latencies are only affected by one actual network delay for large messages (see Table 5.3). In addition, we see that even though HotStuff-2's latency depends only on real communication delays, HotStuff-2 sports lower latency (around 20%) than AlterBFT only up to 8 KB blocks. The reason is that the actual network delay, δ_L, increases as the block size increases. Since the latency of the pipelined version of HotStuff-2 requires three such delays (see Table 5.3), the overall latency grows. As a result, HotStuff-2 achieves latency from 1.7× to 7× higher than AlterBFT when the block size is greater than 8 KB.

Lastly, latencies of FastAlterBFT and partially synchronous protocols rely only on the actual message network delays. Note here that FastAlterBFT requires one voting phase where it needs to receive the votes from all replicas, while Tendermint and HotStuff-2 use two voting phases where they need to receive votes from more than 2/3 of replicas. Consequently, FastAlterBFT optimization works only in optimistic conditions when there are no failures.

In failure-free cases, FastAlterBFT's latency is 1.6 to 2.5× lower than the latency of AlterBFT and almost identical to Tendermint's latency. This result suggests that, in our wide-area setup, a voting phase where the leader needs to receive votes from all replicas requires a similar amount of time as two voting phases with two-third majority quorums.

Throughput

Sync HotStuff, AlterBFT, and FastAlterBFT have similar throughput (see Figure 5.8), as they share the same communication pattern in the failure-free case. Namely, all protocols start working on the next block as soon as they receive the certificate for the previous block.

All three protocols outperform partially synchronous protocols for all system and block sizes, achieving throughput that is 1.4× to 2× higher than Tendermint's. This advantage is due to Tendermint's lack of pipelining. Moreover, they also perform better than HotStuff-2, a partially synchronous protocol with pipelining, by a factor of 1.3× to 7.2×.

Although HotStuff-2 uses pipelining, it only performs better (1.4×) than Tendermint when the block size is up to 8 KB. With larger blocks, HotStuff-2's throughput decreases, becoming worse than Tendermint's. This is attributed to the fact that as block size increases, the real network delay of messages carrying blocks also increases and varies more. Consequently, since HotStuff-2 uses a linear communication pattern where replicas can receive the proposal only from the leader, this can take longer, reducing overall throughput as block size increases.

Lastly, we highlight that even with block sizes up to 8 KB, the throughput of AlterBFT is still around 1.4× better than HotStuff-2's. This is because, although both protocols start ordering the next block after collecting a certificate for the previous block, the certificate in HotStuff-2 requires votes from more than a two-thirds majority of replicas, whereas, in AlterBFT, votes from a simple majority are sufficient.

5.4.4 Performance under attack

We now evaluate AlterBFT and FastAlterBFT under equivocation attack. Contrary to the equivocation attack presented in 4.3.2, the Byzantine replicas are not colluded. Namely, the Byzantine leader of an epoch sends one proposal to half of the replicas and another proposal to the other half, with Byzantine replicas voting for both proposals. Figure 5.9 presents the data for a system of 25

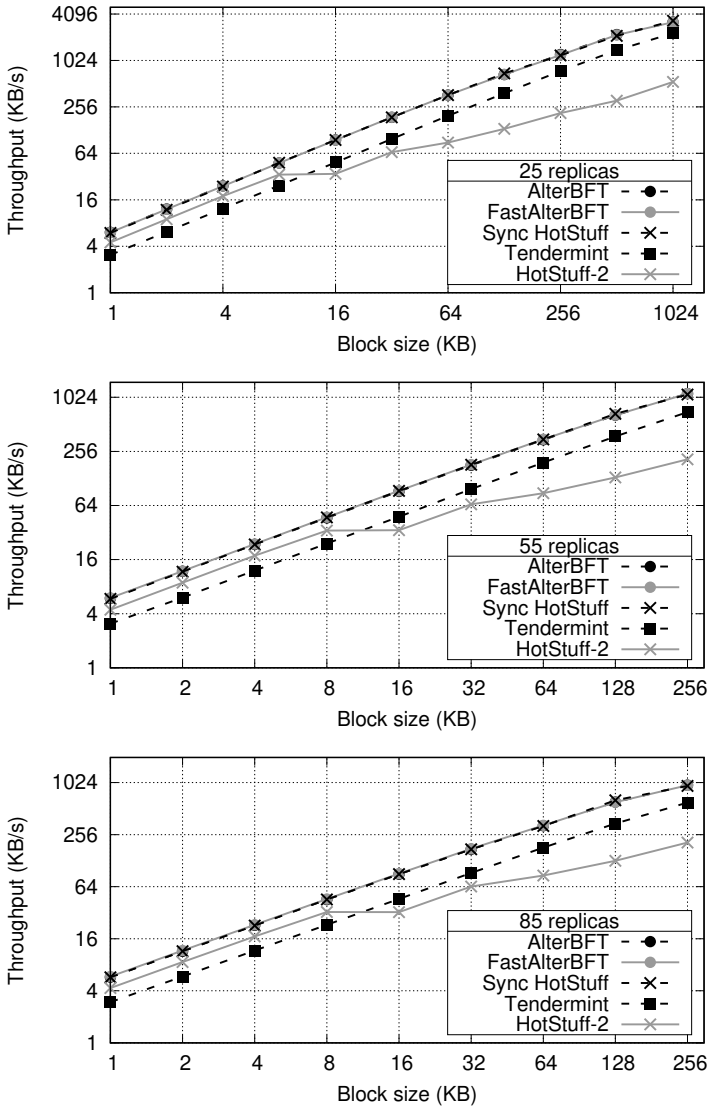

Figure 5.8. Throughput comparison of all protocols with varying system sizes and block sizes (all graphs in log scale).

replicas with 128 KB blocks, varying the number of Byzantine replicas from 2 to 12.

FastAlterBFT's throughput is minimally affected by equivocation attacks due to its chaining mechanism. Honest replicas will not commit a block in epochs with a Byzantine leader, but if they gather a certificate for one of the two proposed blocks, the leader in subsequent epochs (if honest) will extend and indirectly commit one of these blocks. Since in FastAlterBFT, replicas wait for $2\Delta_S$ after

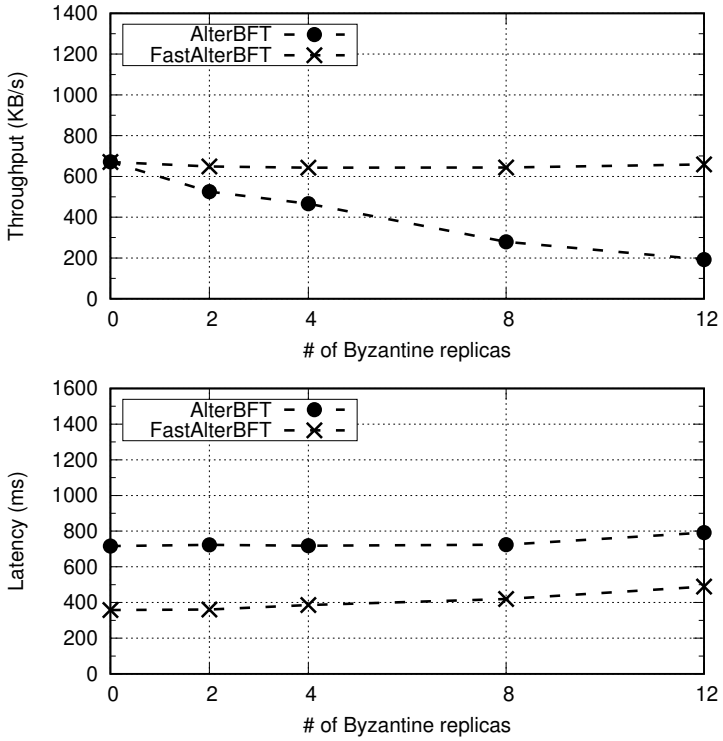

Figure 5.9. AlterBFT and FastAlterBFT throughput (top) and latency (bottom) under equivocation attack, 25 replicas and 128 KB blocks.

receiving an equivocation certificate, they always receive a certificate for one of the blocks before moving to the next epoch. As a result, the throughput is almost identical to that in the failure-free case.

In contrast, AlterBFT replicas move to the next epoch immediately after receiving the equivocation certificate. Consequently, blocks proposed by Byzantine leaders are often wasted. As the number of Byzantine replicas increases, more epochs are wasted, leading to a decrease in AlterBFT's overall throughput.

Moreover, the equivocation attack does not significantly affect the latency of the protocols. With 2 Byzantine replicas, the latencies remain the same. As the number of faulty replicas increases to 4, 8, and 12, the latency of AlterBFT increases by 2%, 2%, and 7.5%, respectively, while the latency of FastAlterBFT increases by 6.5%, 15.7%, and 33%. The increase is due to blocks proposed by Byzantine replicas not being committed in the epochs in which they were proposed but in the first epoch with an honest leader. Since FastAlterBFT has more such blocks than AlterBFT, the impact on average latency is more significant.

Lastly, we observe that the extra $2\Delta_S$ delay in FastAlterBFT has an overall

positive effect on performance during equivocation attacks. Together with the benefits presented in the failure-free case (see Section 5.4.3), this result serves as a compelling argument for the adoption of FastAlterBFT.

5.4.5 Design alternatives

In this section, we evaluate possible alternatives for the hybrid model and AlterBFT. We consider two approaches for synchronous consensus protocols that build on the fact that small messages have reduced and more stable communication delays than large messages (Section 5.2.2):

1. **Limiting message size to a few thousand bytes (i.e., small messages).** In this case, the synchrony bound only needs to accommodate small messages, but this limits the block size to what can fit within a small message. Consequently, multiple consensus instances are required to order blocks larger than the chosen message size.

2. **Sending every large message as many small messages.** A large block can use a single instance of consensus in this case, but a replica can only act on a large block after it has received all smaller messages that correspond to the original block.

We evaluate the first alternative approach and compare it to AlterBFT and Sync HotStuff, where large blocks require conservative synchrony bounds. Specifically, we measure the throughput and latency of Sync HotStuff, where to order a 128 KB block, the leader uses 64 consensus instances. In each instance, the leader proposes a 2 KB chunk (Chunked-HS). We compare this to Sync HotStuff, where a leader uses one consensus instance but sets a conservative synchrony bound (Sync HotStuff).

In these experiments, we use the original Sync HotStuff [3] with a stable leader since it is unclear how the technique could be used with a rotating leader (i.e., how would every leader know which block chunk to propose?). Figure 5.10 shows results for 25 replicas.

Chunked-HS performs worse than Sync HotStuff with a conservative bound: it has 2× higher latency and 55× lower throughput. The reason behind this lies in the overhead of additional consensus executions. Even though Sync HotStuff starts multiple instances in parallel, it cannot start the next instance before certifying the proposal of the current instance, requiring two communication steps before starting a new instance.

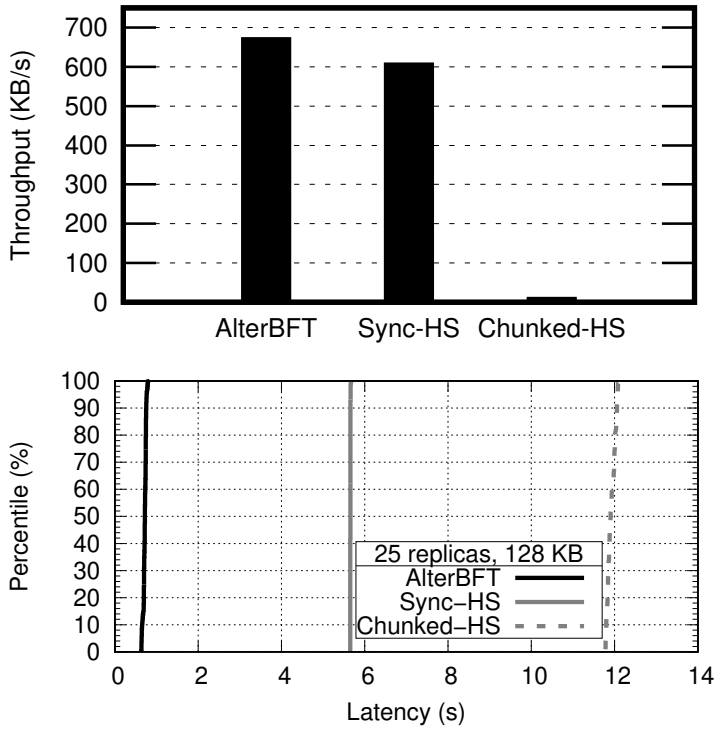

Figure 5.10. Performance comparison of synchronous consensus with chunked proposals (Chunked-HS), conservative bounds (Sync-HS), and AlterBFT for 25 replicas with 128 KB blocks.

In conclusion, empirical evidence suggests that consensus protocols are better off combining small and large messages, instead of resorting to small messages only.

To evaluate the second alternative approach described above, we repeated the same experiments used in Section 5.2.1 to collect message delays. Instead of sending one large message, we divided the messages into small messages and measured the time needed for those small messages to reach their destination and for a response to come back (i.e., round-trip time).

Figure 5.11 compares message delays between N. Virginia and S. Paulo when sending one 128 KB message as a whole (Non-Chopped) versus sending 64 2 KB messages (Chopped). We can see that the delays observed are almost identical.

We conclude that chopping large messages into small messages does not reduce communication delays. Therefore, a large message chopped into small messages is subject to the same timeouts as large messages.

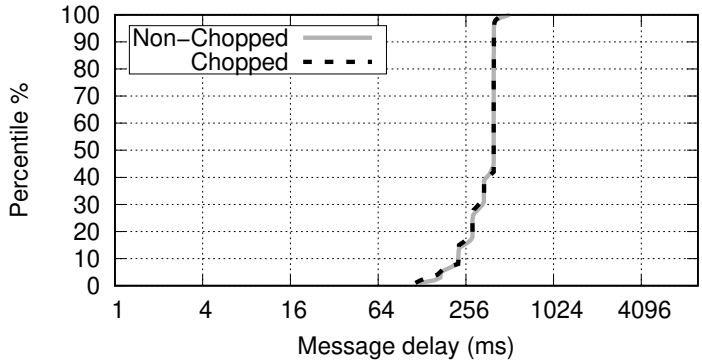

Figure 5.11. Message delays between N. Virginia and S. Paulo (x-axis in log scale) when sending 128 KB messages (Non-Chopped) versus sending 64 2 KB messages (Chopped).

5.4.6 Summary

In this section, we summarize the main takeaways from our performance evaluation.

- *Message size has an important impact on communication delays.* Our empirical data shows that messages below 4KB tend to have low and stable delays, while larger messages experience increasing delays and jitter.

- *AlterBFT significantly outperforms state-of-the-art synchronous protocols.* Depending on block size, AlterBFT's latency is 1.5× to 14.9× lower than Sync HotStuff's latency while maintaining similar throughput.

- *AlterBFT exhibits higher resilience, higher throughput, and comparable latency to partially synchronous protocols.* With 85 replicas, AlterBFT tolerates 15 more faulty replicas, achieves throughput 1.4× to 5× higher, and maintains latency below 1 second.

- *FastAlterBFT optimization enhances AlterBFT's performance both in failure-free scenarios and under equivocation attacks.*

- *Splitting and chopping large blocks does not provide any performance advantage.*

5.5 Conclusion

In this chapter, we introduced the hybrid synchronous system model and AlterBFT, a novel BFT hybrid synchronous consensus protocol. The hybrid synchronous system model differentiates between small messages, which adhere to strict timing bounds, and large messages, which may exceed these bounds but are eventually delivered. AlterBFT's agreement mechanism depends exclusively on the timely delivery of small messages, while the progress of the protocol relies on the eventual delivery of large messages carrying blocks. Consequently, AlterBFT's latency is not affected by the substantial conservative synchrony bounds required for large messages, resulting in a more efficient protocol. AlterBFT demonstrates superior performance compared to traditional synchronous consensus protocols by offering equivalent resilience. Additionally, it surpasses partially synchronous protocols by providing higher throughput and resilience.

Chapter 6

Related work

In this chapter, we review the existing literature relevant to our work, focusing on the key developments and innovations in gossip communication (Section 6.1) and consensus algorithms (Section 6.2). We also discuss additional related proposals (Section 6.3).

6.1 Gossip communication

Gossip algorithms were first introduced by Demers et al. [35] to manage replica consistency in the Xerox Clearinghouse Service [95]. The proposed algorithms were specific for the dissemination of database updates, assumed to not be very frequent (a few per second, at most). The adoption of gossip mechanisms as a building block for the dissemination of arbitrary application messages derives from Bimodal Multicast [15]. The algorithm consists of two phases. In the first phase, messages are disseminated in a best-effort fashion through multicast trees, using IP-multicast when available. In the second phase, processes periodically send to a random-selected peer a list of recently received messages, so that to retransmit, on demand, messages that have not yet been received by the peer. Since then, multiple approaches have been proposed to improve throughput and security of gossip dissemination [16; 40; 56; 66; 76; 79; 86; 113].

Research in gossip-based broadcast algorithms has focused essentially on two issues. First, the efficient dissemination of messages in large-scale systems through the adoption of overlay networks. Proposed approaches consider building pseudo-random network overlays, by selecting links based on geographic proximity and available bandwidth [66; 86], or topological and connectivity properties [76; 79; 114]. A second research direction addresses the cost/effectiveness of epidemic mechanisms which enable processes to request messages that they failed to re-

ceive. The efficiency of these anti-entropy [15; 35] or gossip repair [16; 40; 56; 113] mechanisms is crucial to improve the reliability of gossip dissemination. Efforts have also been made to develop gossip-based services to support large-scale broadcast and multicast algorithms, such as failure detection [109], group membership [47; 63], monitoring and management systems [108].

Semantic Gossip differs from existing approaches because it is designed to support distributed applications that, by themselves, include layers of redundancy. This is the case of Paxos, which includes both typical broadcast steps (to propose values) and the exchange of control messages to ensure agreement, which is a strong form of reliability.

Probabilistic Atomic Broadcast [41] is the algorithm whose behavior most resembles the operation of Paxos atop gossip. The algorithm proceeds in rounds, in each round a process can broadcast a message and should vote for a message, either broadcast or received during the round. Processes periodically exchange the list of messages and associated votes with a random subset of peers. When the number of votes reaches a threshold, all messages in the list are delivered, and the process proceeds to the next round. As in our Paxos deployment, processes send and forward values (broadcast messages) and votes to peers via gossip. Unlike Paxos, the algorithm of [41] only provides probabilistic safety guarantees: two processes may deliver messages in distinct orders, which is equivalent in Paxos to deciding different values in the same consensus instance.

Even though most work on gossip has considered crash failures, recent Byzantine fault-tolerant consensus protocols for large-scale environments (e.g., blockchain) have considered the use of gossip as underlying communication substrate. Tendermint is a blockchain middleware based on a BFT consensus algorithm [20] designed for gossip communication. Tendermint has its own gossip layer implementation, that is application-specific and tightly coupled with the consensus implementation. Casper [21], the BFT consensus algorithm proposed to replace the proof-of-work core of the Ethereum blockchain is also designed for a gossip-based environment. HotStuff [118], the BFT consensus protocol at the core of the Libra Blockchain [8], although not designed for gossip-based communication, considers its adoption as the number of processes participating on consensus (validator nodes) grows [106]. The key architectural aspect that distinguishes these proposals from Semantic Gossip is that gossip in blockchain systems is intertwined with consensus logic. Semantic Gossip exploits application (i.e., consensus) semantics without giving up modularity.

6.2 Consensus

This section surveys the literature on consensus protocols, covering both crash fault-tolerant (CFT) and Byzantine fault-tolerant (BFT) models.

6.2.1 Crash fault-tolerant consensus

There are numerous consensus protocols within the CFT system model [34]. In the following discussion, we focus on Paxos-based protocols and their relevance for Gossip Paxos (Chapter 3).

Paxos is arguably the best-known consensus protocol, widely adopted in both academic and industrial settings [13; 26; 67; 72; 84; 85]. Although it is optimal in terms of communication steps and the number of tolerated failures [73], Paxos is notoriously difficult to understand [18; 94] and implement [26; 67]. Additionally, the Paxos coordinator's distinguished role often becomes a bottleneck in the protocol, limiting performance [13; 72; 83].

Raft [94] is a protocol inspired by Paxos, designed to be easier to understand and implement than Paxos. Raft focuses on replicating a totally ordered sequence of values, rather than solving single instances of consensus. This led to important improvements in the leader-replacement mechanism, used in case of (suspicion of) failures. In the absence of failures, however, the operation of Raft and Paxos are identical [61]: the leader broadcasts values that must be acknowledged by a majority of processes. This makes the semantic extensions proposed for the regular operation of Paxos easily applicable to a gossip-based Raft deployment.

Several Paxos variants address the performance bottlenecks of Paxos. In Mencius [83] processes take turns as the coordinators of successive instances of consensus. While this strategy allows distributing the coordinator load among multiple processes, it does not necessarily improve performance, as it will be ultimately dictated by the slowest coordinator. In S-Paxos [13], the dissemination and ordering of values are detached. Processes disseminate values without the intervention of the coordinator, which proposes value ids in Paxos instances, thus alleviating the coordinator's load. S-Paxos is a good candidate for a gossip-based implementation, where values are inherently disseminated to all processes, while the proposed semantic techniques can be adopted to improve the ordering layer.

Fast Paxos [72] enables any process to propose values directly to all processes, thus bypassing the coordinator. This allows reaching consensus in two communication steps (while Paxos requires three) in instances in which conflicting proposals do not collide. Collisions occur when values are received in distinct orders by processes, which tends to be common with the latency variability of WAN setups,

and can be worsened when communication takes place via gossip. Generalized consensus [71] allows processes to deliver some values, considered independent by the application, in distinct order: the total ordering is relaxed to a partial ordering. The best-known implementation of this approach is EPaxos [90], which allows values to be ordered in two communication steps when no dependent values are concurrently proposed. However, when dependent values are concurrently proposed, EPaxos requires a complex collision-solving procedure, with a communication pattern that is not efficiently implemented atop gossip.

6.2.2 Byzantine fault-tolerant consensus

In this section, we provide a survey of BFT consensus algorithms, a key concept in Chapters 4 and 5. Primarily, we focus on protocols that are designed for a blockchain context.

Early blockchain systems used proof-of-work protocols (e.g., [91; 115]), which do not require knowledge of the full system membership, but consume a lot of energy and have poor performance. In this section, we review protocols that, like BoundBFT and AlterBFT, are membership-based.

Asynchronous protocols

HoneyBadgerBFT [87] is the first practical purely asynchronous consensus protocol designed for a blockchain environment. As such, it makes no assumptions about message bounds Δ and can provide deterministic safety and probabilistic liveness [43]. This protocol represents a more efficient version of the asynchronous atomic broadcast protocol presented in [22]. It consists of a dissemination phase (optimized with the use of erasure codes and Merkle trees), where all processes reliably broadcast transactions in parallel, and agreement phase that consists of n concurrent asynchronous binary Byzantine agreement (ABBA). Later, in [53; 54] agreement phase was improved by replacing n ABBAs with a single asynchronous multi-value validated Byzantine agreement (MVBA). This resulted in better performance, both throughput and latency. Lastly, the performance was further improved by decoupling transaction dissemination and agreement [117] and further by allowing them to be executed completely concurrently [48]. The main advantage of all these protocols is that they can achieve termination without relying on synchrony. Consequently, they can order blocks in asynchronous periods where the performance of deterministic partially synchronous or synchronous solutions can deteriorate [87]. However, in good-case scenarios, they perform worse than partially synchronous and synchronous protocols,

because they require more communication steps. As a consequence, some protocols use a leader-based partially synchronous protocol to improve the latency in good cases [49; 69; 81; 98].

Partially synchronous protocols

Dwork, Lynch, and Stockmeyer introduced the partially synchronous system model [37], defining a degree of synchrony sufficient to circumvent the FLP impossibility result [43] and achieve deterministic consensus.

The first practical BFT consensus protocol designed for a partially synchronous system model is PBFT [24], a leader-based protocol that can commit a value in three communication steps. PBFT was originally built for scenarios with a stable leader, where the leader change occurs only in the event of a leader failure. Consequently, the leader replacement mechanism is costly, incurring $O(n^3)$ communication overhead.

In contrast, blockchain systems require that every node be given the opportunity to act as a leader. This is essential to ensure that each node has a fair chance to propose a block and collect the associated fees, while also preventing a Byzantine leader from indefinitely censoring transactions. Consequently, a new line of BFT protocols tailored for blockchain contexts has emerged, focusing on facilitating efficient and frequent leader changes.

Tendermint [20] removes the need for a leader replacement sub-protocol, incorporating leader changes as part of its regular execution. It operates in rounds, with each round having a dedicated leader and a failure-free communication pattern similar to PBFT's. Unlike PBFT, the new leader in a given round r (a concept akin to a view in PBFT) does not need to execute a leader replacement sub-protocol to determine the value to propose. Instead, each process maintains the most up-to-date value in a local state, $validValue$, and proposes this value when leader in a round. The mechanism ensuring that a value proposed by an honest process is eventually accepted by all honest processes—and thus decided—relies on gossip communication (which ensures that after GST, all honest processes witness the same set of messages) and a round transition mechanism (which ensures that after GST, an honest process does not advance to the next round without learning the most up-to-date value). In BoundBFT and AlterBFT, we adopt a similar design.

HotStuff [119] designed a leader rotation mechanism that requires linear communication $O(n)$ and is responsive, meaning that a new leader needs to wait for only $n - f$ messages before proposing a value, rather than the maximum network delay. HotStuff achieves responsiveness at the cost of additional com-

munication, requiring three voting phases compared to the two phases needed in PBFT and Tendermint. HotStuff-2 [82] demonstrates that the additional phase in HotStuff is unnecessary in practice and achieves responsiveness without extra communication under optimistic conditions, when there is a sequence of honest leaders and a synchronous network.

An alternative approach to prevent censorship and allow multiple leaders is to enable parallel leaders [30; 88; 103]. These solutions provide higher throughput but are more complex as they must address challenges such as request duplication in the context of parallel leaders [103]. Recent work has shown how a leader-driven protocol can be transformed into a scalable multi-leader one using a new primitive called Sequenced (Total Order) Broadcast [104].

Synchronous protocols

Synchronous BFT consensus protocols require a majority of honest replicas [42; 44; 64], as opposed to partially synchronous and asynchronous protocols, which require a two-third majority. Dfinity [58] is the first synchronous consensus designed for blockchains. Contrary to the early BFT protocols in the synchronous model [36; 74], Dfinity does not assume lock-step execution where replicas execute the protocol in rounds and messages sent at the start of the round arrive by the end of the round. Instead, it assumes that replicas start the protocol within Δ time. Dfinity's throughput is affected by the maximum network delay Δ because every replica at the beginning of each round waits for 2Δ before casting a vote.

Abraham et al. [3] introduced Sync HotStuff, which removes the effect of maximum network delay on throughput, achieving throughput comparable to the partially synchronous HotStuff, and also reducing latency. A rotating-leader version of Sync HotStuff was introduced in [5]. AlterBFT and BoundBFT share similar common-case behavior as rotating-leader Sync HotStuff. However, they have different epoch synchronization mechanisms. Moreover, AlterBFT is designed for hybrid synchronous model and its agreement does not require timely delivery of all messages.

Thunderella [96] points out that the latency of synchronous BFT consensus protocols does not need to depend on Δ when the actual number of faults is less than $1/4$ of the replicas. They refer to these protocols as *optimistically responsive* since their latency does not depend on Δ only in some special, optimistic, conditions. FastAlterBFT is optimistically responsive when there are no failures in the system [4; 52; 68].

Protocols based on extended hardware

Some protocols increase resilience by relying on trusted components. The main idea is to execute key functionality, such as appending to a log [27] or incrementing a counter [77], inside a trusted execution environment (e.g., Intel SGX enclaves [100]). Extended hardware has been used to allow both PBFT [27; 77; 93; 111] and HotStuff [33; 116] to tolerate a minority of Byzantine replicas. BoundBFT and AlterBFT do not require any trusted components and rely on synchrony instead.

Another approach is to divide the system into two parts [110]: a synchronous subsystem that transmits control messages, and an asynchronous subsystem that transmits the payload. This model was generalized to the wormhole hybrid distributed system model where secure and timely components co-exist [29; 112]. AlterBFT also differentiates between two types of messages, but does not assume the existence of any separate subsystem or special components.

DAG-based protocols

HashGraph [12] introduced the idea of building a directed acyclic graph (DAG) of messages and designing an algorithm that will solve BFT consensus just by interpreting the DAG without sending any additional messages. Aleph [46] improved the DAG structure by adding rounds, and a round version of the DAG was efficiently implemented in Narhwal [32]. Different versions of DAG-based BFT consensus protocols that built on Narhwal's DAG have been proposed for both asynchronous [32; 65; 101] and partially synchronous system models [102; 105]. More recently, the authors of [11] demonstrated how to further reduce the latency of DAG-based protocols by eliminating the requirement for nodes in the DAG to be certified. All these systems tolerate fewer than 1/3 of Byzantine replicas. Designing a synchronous DAG-based protocol that can tolerate a minority of Byzantine replicas is still an open question.

6.3 Additional proposals

Guo et al. [55] introduced the "weak synchronous model" (called mobile sluggish model in [3] for consistency with other works in the literature). The model tolerates Byzantine replicas and allows some honest replicas to be slow, that is, the messages received from or sent by slow processes can violate synchrony bounds. However, this is true only in situations when the actual number of faulty and slow processes is a minority at any point in time during the execution. The

first BFT consensus protocol presented in the weak synchronous model was PiLi [25], with latency between 40Δ and 65Δ. In [3], the authors showed how Sync HotStuff can be adapted to the weak synchronous model. One open question is how our hybrid model could be combined with the weak synchronous model, yielding a version of AlterBFT whose safety would rely on the timely delivery of small messages between the majority of timely and honest replicas.

XFT [80] is based on the observation that typical BFT consensus protocols assume a powerful adversary that fully controls malicious processes and the network between honest processes. They note that this type of adversary is unrealistic. We share this view, and consider an adversary that cannot control the network between honest processes in Chapters 4 and 5. XFT differentiates between three types of faulty processes: crash, Byzantine, and partitioned (i.e., processes that cannot exchange messages with other honest processes within the known synchrony bound). It ensures progress as long as the total number of faulty processes in the system is lower than $f < n/2$. In other words, XFT assumes a majority of honest replicas that can communicate timely. Since selecting a quorum of $f + 1$ responsive replicas out of n replicas requires an exponential number of attempts, the solution is practical when f is small.

The hybrid fault model introduced in [107] distinguishes between different types of failures and proposes different thresholds for crash and Byzantine failures. Its most recent refinement [97] expands the work by adding the threshold for slow replicas. This approach allowed the design of more cost-efficient (tolerating the same number of failures with fewer replicas) protocols in the data center environment.

Gilad et al. [50] propose a model with different timing assumptions for progress and agreement. Namely, for progress, they assume that most (e.g., 95%) processes can send messages that will be received by most (e.g., 95%) other processes within a known time-bound. For agreement, they assume a network can be asynchronous for a long but bounded time (e.g., at most one day or week). Afterwards, the network must be strongly synchronous for a reasonably long period (e.g., a few hours or a day). Furthermore, they assume loosely synchronous clocks to recover progress after asynchronous periods. Their consensus algorithm can tolerate the same number of Byzantine replicas as asynchronous but provide probabilistic guarantees.

The authors in [31] proposed a model where the replicas are equipped with hardware clocks with a bounded drift rate. Moreover, they design protocols by leveraging the hardware clocks to detect untimely events accurately. These protocols don't have a strict bound on the number of synchronous/asynchronous and correct/faulty replicas and do not consider Byzantine failures.

The transmission fault model [14; 89] approaches the system from a different angle. Instead of modeling faulty replicas or links, they assume that messages can be arbitrarily lost or corrupted. The common aspect between this model and ours is that both focus on messages. However, since the blockchain environment usually comes with the PKI infrastructure and every message is usually signed, corrupted messages are easily detectable.

Aguilera et al. [7] examine how to enhance the performance of consensus algorithms in synchronous systems with crash failures by utilizing fast failure detectors. These detectors are implemented through specialized hardware or expedited message delivery mechanisms. Expedited message delivery prioritizes critical messages by tagging them for faster processing in network queues or using a separate transmission medium. Some papers demonstrate how expedited messages can be enforced in real-time systems [59; 60]. This approach can further improve the performance of AlterBFT by marking type $\mathscr{S}$ messages as expedited.

Chapter 7

Concluding remarks

This thesis has significantly advanced the understanding and development of consensus protocols, which are vital for the reliability and fault tolerance of distributed systems. By tackling critical challenges posed by modern decentralized environments (e.g., blockchain systems), this research provides novel insights and practical solutions to enhance consensus mechanisms. The primary focus was on three key areas: integrating gossip-based communication with consensus protocols, assessing the robustness of synchronous Byzantine fault-tolerant (BFT) protocols under synchrony violations, and developing a hybrid synchrony model to improve performance. Through comprehensive theoretical analysis and rigorous experimental evaluation, this work has not only addressed current limitations but also set the stage for future innovations in the field of distributed systems.

7.1 Research assessment

This section summarizes the key contributions of this thesis.

Gossip-based consensus protocols. This thesis has explored the deployment of consensus protocols in partially connected networks utilizing gossip communication. Our investigation into the Paxos consensus algorithm revealed significant latency and throughput overheads caused by gossip-based communication. To mitigate these issues, we introduced Semantic Gossip, which employs semantic filtering and semantic aggregation. Our experimental evaluation confirmed that Semantic Gossip significantly reduces message overhead, enhances performance, and maintains the reliability of gossip communication despite message

loss. These results indicate that similar optimizations could benefit other agreement protocols, suggesting promising directions for future research and practical applications.

Robustness of synchronous consensus. This thesis presents a novel approach for assessing the resilience of synchronous BFT consensus protocols in the face of synchrony violations, considering scenarios with and without Byzantine replicas. Applied to BoundBFT, a novel BFT synchronous consensus protocol, this approach demonstrates that BoundBFT can withstand synchrony violations without compromising correctness, leveraging communication diversity and redundancy. Our experimental results show that BoundBFT achieves lower synchrony bounds and improved performance compared to existing protocols. These findings underscore the effectiveness of the proposed assessment approach and offer valuable insights into the robustness of synchronous consensus protocols under challenging conditions.

Hybrid synchronous system model and AlterBFT. This thesis has presented a three-month experimental study on communication delays in a geographically distributed system. Based on the collected insights, we proposed the hybrid synchronous system model, which reflects the observed data more accurately than the classical synchronous model by distinguishing between small messages with strict timing bounds and large messages with eventual delivery.

Using this model, we designed AlterBFT, a new BFT consensus protocol. Our experimental evaluation showed that AlterBFT relies on the timely delivery of small messages for agreement while allowing progress with the eventual delivery of large messages. AlterBFT achieves the same fault tolerance as synchronous protocols, with up to 15× lower latency and similar throughput. Additionally, AlterBFT offers higher throughput and comparable latency to partially synchronous protocols. These results highlight the effectiveness of the hybrid synchronous model and suggest that AlterBFT can provide significant improvements in practical applications.

7.2 Future directions

The insights and contributions from this thesis pave the way for several future research directions:

7.2.1 Byzantine Gossip Consensus

We plan to investigate the applicability of Semantic Gossip to BFT consensus protocols. In particular, we have started exploring the integration of Semantic Gossip with Tendermint, a well-known BFT consensus protocol that already utilizes gossip-based communication. This integration aims to reduce message overhead while maintaining security guarantees in adversarial settings.

However, adapting semantic aggregation for BFT protocols requires special care to prevent new attack vectors. Aggregated messages must include signed votes to ensure their validity and resist malicious behaviors. More efficient cryptographic schemes, such as Boneh-Lynn-Shacham (BLS) signatures [19], could be employed to reduce message size while preserving security.

Our preliminary experiments with Tendermint have shown promising results, indicating that Semantic Gossip can further enhance the performance and scalability of BFT protocols. Future research will focus on refining these techniques and conducting extensive evaluations in various network conditions.

7.2.2 Hybrid model validation

In Chapter 5, we introduced the new hybrid synchronous system model, motivated by data on message delays between replicas located in five AWS and five DigitalOcean regions. The replicas were hosted on medium-sized machines and communicated via TCP. To further substantiate the viability of this model, we plan to expand our data collection and analysis. Specifically, we aim to:

1. Include additional geographic regions to provide a more comprehensive understanding of global communication delays.

2. Utilize various data transfer protocols, such as QUIC, to evaluate their impact on communication latency and reliability.

3. Expand our experiments to include additional cloud service providers beyond AWS and DigitalOcean, such as CloudLab, Microsoft Azure, and Google Cloud, to assess the stability of the hybrid synchronous model across diverse infrastructure environments.

4. Experiment with different VM instance types, including larger machines, to determine the impact of varying bandwidth availability on message delays.

By broadening the scope of our experimental study, we aim to validate the hybrid synchronous model's applicability and effectiveness in a wider range of practical scenarios.

7.2.3 AlterBRB

We intend to design AlterBRB, a new BFT reliable broadcast protocol for the hybrid system model. Recent research [17; 28; 51] has shown the importance of such protocols in building decentralized online payment systems [91] that do not require consensus. We believe AlterBRB may serve as an alternative to asynchronous Byzantine reliable broadcast protocols used in these systems, potentially offering comparable performance while providing higher resilience and availability.

Reliable dissemination of data and ordering on hashes/metadata is a known technique [38] for increasing the performance of an ordering service. This technique is also utilized in modern blockchain systems [32; 48; 117]. AlterBRB can be employed as an efficient reliable broadcast protocol and combined with AlterBFT or other synchronous protocols to create an efficient ordering service. This service would tolerate a minority of Byzantine processes, unlike current partially synchronous solutions, which tolerate only up to one-third of malicious processes.

7.2.4 Handling large blocks in partially synchronous protocols

In our evaluation in Chapter 5, we observed that while partially synchronous systems do not rely on conservative Δ values, their latency tends to increase with larger block sizes. Specifically, in a system with 25 replicas, the latency of HotStuff-2 increased by 10× when the block size was increased from 1KB to 1MB. Similarly, Tendermint's latency increased by 1.4×. We believe this is because the actual communication delays for large messages are significantly higher compared to small messages.

To address this issue, we propose modifying these protocols to alleviate the latency increase associated with larger block sizes. Our approach involves sending the message containing the block in parallel with the voting process. Specifically, a replica would send a propose message carrying a hash of the block and a message with the full block in parallel. Replicas would vote after receiving the hash of the block, even if they have not yet received the full block. However, replicas would lock on the block and send a second phase vote message only after receiving the full block. This approach allows the protocol to proceed with the first voting phase without waiting for the full block to be received, thereby giving more time for block propagation and potentially reducing overall latency.

Bibliography

[1] Abraham, I., Gueta, G., Malkhi, D., Alvisi, L., Kotla, R. and Martin, J.-P. [2017]. Revisiting fast practical byzantine fault tolerance.

[2] Abraham, I., Gueta, G., Malkhi, D. and Martin, J.-P. [2018]. Revisiting fast practical byzantine fault tolerance: Thelma, velma, and zelma.

[3] Abraham, I., Malkhi, D., Nayak, K., Ren, L. and Yin, M. [2020]. Sync HotStuff: Simple and practical synchronous state machine replication, *2020 IEEE Symposium on Security and Privacy (SP)*.

[4] Abraham, I., Nayak, K., Ren, L. and Shrestha, N. [2020]. On the optimality of optimistic responsiveness, Cryptology ePrint Archive, Paper 2020/458. `https://eprint.iacr.org/2020/458`.
URL: `https://eprint.iacr.org/2020/458`

[5] Abraham, I., Nayak, K. and Shrestha, N. [2022]. Optimal good-case latency for rotating leader synchronous BFT, *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, pp. 27:1–27:19.

[6] Aguilera, M. K., Chen, W. and Toueg, S. [1999]. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks, *Theoretical Computer Science* **220**(1): 3–30.

[7] Aguilera, M. K., Lann, G. L. and Toueg, S. [2002]. On the impact of fast failure detectors on real-time fault-tolerant systems, *International Symposium on Distributed Computing*.
URL: `https://api.semanticscholar.org/CorpusID:15584713`

[8] Amsden, Z., Arora, R., Bano, S., Baudet, M. et al. [2020]. The libra blockchain, *White paper*, The Libra Association. [Accessed 2020-06-01].
URL: `https://developers.libra.org/docs/the-libra-blockchain-paper`

[9] Androulaki, E., Barger, A., Bortnikov, V., Cachin, C., Christidis, K., De Caro, A., Enyeart, D., Ferris, C., Laventman, G., Manevich, Y., Muralidharan, S., Murthy, C., Nguyen, B., Sethi, M., Singh, G., Smith, K., Sorniotti, A., Stathakopoulou, C., Vukolić, M., Cocco, S. W. and Yellick, J. [2018]. Hyperledger fabric: A distributed operating system for permissioned blockchains, *Proceedings of the Thirteenth EuroSys Conference*, pp. 1–15.

[10] Aublin, P.-L., Guerraoui, R., Knežević, N., Quéma, V. and Vukolić, M. [2015]. The next 700 bft protocols, *ACM Trans. Comput. Syst.* **32**(4).
URL: https://doi.org/10.1145/2658994

[11] Babel, K., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Koshy, A., Sonnino, A. and Tian, M. [2024]. Mysticeti: Reaching the limits of latency with uncertified dags.
URL: https://arxiv.org/abs/2310.14821

[12] Baird, L. and Luykx, A. [2020]. The hashgraph protocol: Efficient asynchronous BFT for high-throughput distributed ledgers, *2020 International Conference on Omni-layer Intelligent Systems, COINS 2020, Barcelona, Spain, August 31 - September 2, 2020*, IEEE, pp. 1–7.
URL: https://doi.org/10.1109/COINS49042.2020.9191430

[13] Biely, M., Milosevic, Z., Santos, N. and Schiper, A. [2012]. S-paxos: Offloading the leader for high throughput state machine replication, *2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS'12, pp. 111–120.

[14] Biely, M., Widder, J., Charron-Bost, B., Gaillard, A., Hutle, M. and Schiper, A. [2007]. Tolerating corrupted communication, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, Association for Computing Machinery, New York, NY, USA, pp. 244–253.
URL: https://doi.org/10.1145/1281100.1281136

[15] Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M. and Minsky, Y. [1999]. Bimodal multicast, *ACM Transactions on Computer Systems (TOCS)* **17**(2): 41–88.

[16] Birman, K. P., van Renesse, R. and Vogels, W. [2001]. Spinglass: secure and scalable communication tools for mission-critical computing, *Proceedings DARPA Information Survivability Conference and Exposition II*, Vol. 2 of *DISCEX'01*, pp. 85–99.

[17] Blackshear, S., Chursin, A., Danezis, G., Kichidis, A., Kokoris-Kogias, L., Li, X., Logan, M., Menon, A., Nowacki, T., Sonnino, A., Williams, B. and Zhang, L. [2024]. Sui lutris: A blockchain combining broadcast and consensus.
URL: https://arxiv.org/abs/2310.18042

[18] Boichat, R., Dutta, P., Frølund, S. and Guerraoui, R. [2003]. Deconstructing paxos, *ACM SIGACT News* **34**(1): 47–67.

[19] Boneh, D., Lynn, B. and Shacham, H. [2001]. Short signatures from the weil pairing, *in* C. Boyd (ed.), *Advances in Cryptology — ASIACRYPT 2001*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 514–532.

[20] Buchman, E., Kwon, J. and Milosevic, Z. [2018]. The latest gossip on BFT consensus, arXiv:1807.04938 [cs.DC].
URL: https://arxiv.org/abs/1807.04938

[21] Buterin, V. and Griffith, V. [2017]. Casper the Friendly Finality Gadget.
URL: https://arxiv.org/abs/1710.09437

[22] Cachin, C., Kursawe, K., Petzold, F. and Shoup, V. [2001]. Secure and efficient asynchronous broadcast protocols, Vol. 2139.

[23] Cachin, C., Schubert, S. and Vukolić, M. [2016]. Non-determinism in byzantine fault-tolerant replication.
URL: https://arxiv.org/abs/1603.07351

[24] Castro, M. and Liskov, B. [1999]. Practical Byzantine Fault Tolerance, *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, USENIX Association, USA, pp. 173–186.

[25] Chan, T.-H. H., Pass, R. and Shi, E. [2018]. Pili: An extremely simple synchronous blockchain, Cryptology ePrint Archive, Paper 2018/980. https://eprint.iacr.org/2018/980.
URL: https://eprint.iacr.org/2018/980

[26] Chandra, T. D., Griesemer, R. and Redstone, J. [2007]. Paxos made live, *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC'07, ACM Press, pp. 398–407.

[27] Chun, B.-G., Maniatis, P., Shenker, S. and Kubiatowicz, J. [2007]. Attested append-only memory, *ACM SIGOPS Operating Systems Review* **41**(6): 189–204.

[28] Collins, D., Guerraoui, R., Komatovic, J., Kuznetsov, P., Monti, M., Pavlovic, M., Pignolet, Y.-A., Seredinschi, D.-A., Tonkikh, A. and Xygkis, A. [2020]. Online payments by merely broadcasting messages, *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 26–38.

[29] Correia, M., Neves, N. and Verissimo, P. [2004]. How to tolerate half less one byzantine nodes in practical distributed systems, *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, 2004.*, pp. 174–183.

[30] Crain, T., Natoli, C. and Gramoli, V. [2018]. Evaluating the red belly blockchain.
URL: `https://arxiv.org/abs/1812.11747`

[31] Cristian, F. and Fetzer, C. [1999]. The timed asynchronous distributed system model, *IEEE Transactions on Parallel and Distributed Systems* **10**(6): 642–657.

[32] Danezis, G., Kogias, E. K., Sonnino, A. and Spiegelman, A. [2022]. Narwhal and tusk: A dag-based mempool and efficient bft consensus.

[33] Decouchant, J., Kozhaya, D., Rahli, V. and Yu, J. [2022]. DAMYSUS: streamlined bft consensus leveraging trusted components, *Proceedings of the Seventeenth European Conference on Computer Systems*, ACM.

[34] Défago, X., Schiper, A. and Urbán, P. [2004]. Total order broadcast and multicast algorithms: Taxonomy and survey, *ACM Comput. Surv.* **36**(4): 372–421.
URL: `https://doi.org/10.1145/1041680.1041682`

[35] Demers, A., Greene, D., Hauser, C., Irish, W. and Larson, J. [1987]. Epidemic algorithms for replicated database maintenance, *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC'87, ACM Press, pp. 1–12.

[36] Dolev, D. and Strong, H. [1983]. Authenticated algorithms for byzantine agreement, *SIAM J. Comput.* **12**: 656–666.

[37] Dwork, C., Lynch, N. and Stockmeyer, L. [1988]. Consensus in the presence of partial synchrony, *Journal of the ACM (JACM)* **35**(2): 288–323.

[38] Ekwall, R. and Schiper, A. [2006]. Solving atomic broadcast with indirect consensus, *International Conference on Dependable Systems and Networks (DSN'06)*, pp. 156–165.

[39] Erdös, P. and Kennedy, J. [1987]. k-Connectivity in random graphs, *European Journal of Combinatorics* **8**(3): 281–286.

[40] Eugster, P. T., Guerraoui, R., Handurukande, S. B., Kouznetsov, P. and Kermarrec, A.-M. [2003]. Lightweight probabilistic broadcast, *ACM Transactions on Computer Systems (TOCS)* **21**(4): 341–374.

[41] Felber, P. and Pedone, F. [n.d.]. Probabilistic atomic broadcast, *Proceeedings of 21st IEEE Symposium on Reliable Distributed Systems, 2002*, SRDS '02, pp. 170–179.

[42] Fischer, M. J., Lynch, N. A. and Merritt, M. [1986]. Easy impossibility proofs for distributed consensus problems, *Distrib. Comput.* **1**(1): 26–39.
URL: `https://doi.org/10.1007/BF01843568`

[43] Fischer, M. J., Lynch, N. A. and Paterson, M. S. [1985]. Impossibility of distributed consensus with one faulty process, *Journal of ACM* **32**(2): 374–382.

[44] Fitzi, M. [2003]. *Generalized Communication and Security Models in Byzantine Agreement*, PhD thesis, ETH Zurich. Reprint as vol. 4 of ETH Series in Information Security and Cryptography, ISBN 3-89649-853-3, Hartung-Gorre Verlag, Konstanz, 2003.

[45] Friedman, R. and van Renesse, R. [1995]. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols, *Technical Report 94-1527*, Cornell University, Dept. of Computer Science. Submitted to IEEE Transactions on Networking.

[46] Gagol, A., Lesniak, D., Straszak, D. and Swietek, M. [2019]. Aleph: Efficient atomic broadcast in asynchronous networks with byzantine nodes, *Proceedings of the 1st ACM Conference on Advances in Financial Technologies* .
URL: `https://api.semanticscholar.org/CorpusID:199577506`

[47] Ganesh, A. J., Kermarrec, A.-M. and Massoulié, L. [2003]. Peer-to-peer membership management for gossip-based protocols, *IEEE Transactions on Computers* **52**(2): 139–149.

[48] Gao, Y., Lu, Y., Lu, Z., Tang, Q., Xu, J. and Zhang, Z. [2022]. Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, Association for Computing Machinery, New York, NY, USA, pp. 1187–1201.
URL: https://doi.org/10.1145/3548606.3559379

[49] Gelashvili, R., Kokoris-Kogias, L., Sonnino, A., Spiegelman, A. and Xiang, Z. [2021]. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback.

[50] Gilad, Y., Hemo, R., Micali, S., Vlachos, G. and Zeldovich, N. [2017]. Algorand: Scaling byzantine agreements for cryptocurrencies, *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, pp. 51–68.
URL: https://doi.org/10.1145/3132747.3132757

[51] Guerraoui, R., Kuznetsov, P., Monti, M., Pavlovič, M. and Seredinschi, D.-A. [2019]. The consensus number of a cryptocurrency, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, ACM.
URL: https://doi.org/10.1145%2F3293611.3331589

[52] Gueta, G., Abraham, I., Grossman, S., Malkhi, D., Pinkas, B., Reiter, M., Seredinschi, D.-A., Tamir, O. and Tomescu, A. [2019]. SBFT: A scalable and decentralized trust infrastructure, pp. 568–580.

[53] Guo, B., Lu, Y., Lu, Z., Tang, Q., Xu, J. and Zhang, Z. [2022]. Speeding dumbo: Pushing asynchronous bft closer to practice, Cryptology ePrint Archive, Paper 2022/027. https://eprint.iacr.org/2022/027.
URL: https://eprint.iacr.org/2022/027

[54] Guo, B., Lu, Z., Tang, Q., Xu, J. and Zhang, Z. [2020]. Dumbo: Faster asynchronous bft protocols, *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, CCS '20, Association for Computing Machinery, New York, NY, USA, pp. 803–818.
URL: https://doi.org/10.1145/3372297.3417262

[55] Guo, Y., Pass, R. and Shi, E. [2019]. Synchronous, with a chance of partition tolerance, *Advances in Cryptology – CRYPTO 2019*, pp. 499–529.

[56] Gupta, I., Birman, K. P. and van Renesse, R. [2002]. Fighting fire with fire: using randomized gossip to combat stochastic scalability limits, *Quality and Reliability Engineering International* **18**(3): 165–184.

[57] Hadzilacos, V. and Toueg, S. [1994]. A modular approach to fault-tolerant broadcasts and related problems, *Technical report*, USA.

[58] Hanke, T., Movahedi, M. and Williams, D. [2018]. Dfinity technology overview series, consensus system, arXiv:1805.04548 [cs.DC].

[59] Hermant, J.-F. and Le Lann, G. [2002]. Fast asynchronous uniform consensus in real-time distributed systems, *Computers, IEEE Transactions on* **51**: 931– 944.

[60] Hermant, J.-F. and Widder, J. [2006]. Implementing reliable distributed real-time systems with the θ-model, *in* J. H. Anderson, G. Prencipe and R. Wattenhofer (eds), *Principles of Distributed Systems*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 334–350.

[61] Howard, H., Schwarzkopf, M., Madhavapeddy, A. and Crowcroft, J. [2015]. Raft refloated, *ACM SIGOPS Operating Systems Review* **49**(1): 12–21.

[62] Hubert, B., Maxwell, G., van Oosterhout, M., van Mook, R., Schroeder, P. B. et al. [2002]. Linux advanced routing & traffic control HOWTO, `https://lartc.org/lartc.html`. [Accessed 2020-05-17].

[63] Johansen, H., Allavena, A. and van Renesse, R. [2006]. Fireflies: scalable support for intrusion-tolerant network overlays, *ACM SIGOPS Operating Systems Review* **40**(4): 3–13.

[64] Katz, J. and Koo, C.-Y. [2006]. On expected constant-round protocols for byzantine agreement, *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference*, Vol. 4117 of *Lecture Notes in Computer Science*, Springer, pp. 445–462.
URL: `https://iacr.org/archive/crypto2006/41170440/41170440.pdf`

[65] Keidar, I., Kokoris-Kogias, E., Naor, O. and Spiegelman, A. [2021]. All you need is dag.

[66] Kempe, D., Kleinberg, J. and Demers, A. [2004]. Spatial gossip and resource location protocols, *Journal of the ACM (JACM)* **51**(6): 943–967.

[67] Kirsch, J. and Amir, Y. [2008]. Paxos for system builders, *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS'08, ACM Press.

[68] Kotla, R., Alvisi, L., Dahlin, M., Clement, A. and Wong, E. [2010]. Zyzzyva: Speculative byzantine fault tolerance, *ACM Trans. Comput. Syst.* **27**(4).
URL: https://doi.org/10.1145/1658357.1658358

[69] Kursawe, K. and Shoup, V. [2001]. Optimistic asynchronous atomic broadcast, Cryptology ePrint Archive, Paper 2001/022. https://eprint.iacr.org/2001/022.
URL: https://eprint.iacr.org/2001/022

[70] Lamport, L. [1998]. The part-time parliament, *ACM Transactions on Computer Systems* **16**(2): 133–169.

[71] Lamport, L. [2005]. Generalized consensus and paxos, *Technical Report MSR-TR-2005-33*, Microsoft Research.
URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2005-33.pdf

[72] Lamport, L. [2006a]. Fast Paxos, *Distributed Computing* **19**(2): 79–103.

[73] Lamport, L. [2006b]. Lower bounds for asynchronous consensus, *Distributed Computing* **19**(2): 104–125.

[74] Lamport, L., Shostak, R. and Pease, M. [1982]. The byzantine generals problem, *ACM Trans. Program. Lang. Syst.* **4**(3): 382–401.
URL: https://doi.org/10.1145/357172.357176

[75] Lampson, B. W. [2001]. The abcd's of paxos, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC'01.
URL: https://doi.org/10.1145/383962.383969

[76] Leitao, J., Pereira, J. and Rodrigues, L. [2007]. Epidemic broadcast trees, *26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pp. 301–310.

[77] Levin, D., Douceur, J. R., Lorch, J. R. and Moscibroda, T. [2009]. TrInc: Small trusted hardware for large distributed systems., *NSDI*, Vol. 9, pp. 1–14.

[78] *Libp2p* [n.d.]. https://libp2p.io. [Accessed 2023-01-12].

[79] Lin, M.-J. and Marzullo, K. [1999]. Directional gossip: Gossip in a wide area network, *Proceedings of Third European Dependable Computing Conference*, EDCC-3, Springer Berlin Heidelberg, pp. 364–379.

[80] Liu, S., Viotti, P., Cachin, C., Quéma, V. and Vukolic, M. [2016]. XFT: Practical fault tolerance beyond crashes, *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, USA, pp. 485–500.

[81] Lu, Y., Lu, Z. and Tang, Q. [2021]. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined bft, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* .
URL: https://api.semanticscholar.org/CorpusID:232257767

[82] Malkhi, D. and Nayak, K. [2023]. Extended abstract: Hotstuff-2: Optimal two-phase responsive BFT, *IACR Cryptol. ePrint Arch.* p. 397.
URL: https://eprint.iacr.org/2023/397

[83] Mao, Y., Junqueira, F. P. and Marzullo, K. [2008]. Mencius: Building efficient replicated state machines for wans, *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, USENIX Association, pp. 369–384.

[84] Marandi, P. J., Benz, S., Pedonea, F. and Birman, K. P. [2014]. The performance of paxos in the cloud, *IEEE 33rd International Symposium on Reliable Distributed Systems*, pp. 41–50.

[85] Marandi, P. J., Primi, M., Schiper, N. and Pedone, F. [2010]. Ring Paxos: A high-throughput atomic broadcast protocol, *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, pp. 527–536.

[86] Melamed, R. and Keidar, I. [2004]. Araneola: a scalable reliable multicast system for dynamic environments, *Third IEEE International Symposium on Network Computing and Applications*, NCA 2004.

[87] Miller, A., Xia, Y., Croman, K., Shi, E. and Song, D. [2016]. The honey badger of bft protocols, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, Association for Computing Machinery, New York, NY, USA, pp. 31–42.
URL: https://doi.org/10.1145/2976749.2978399

[88] Milosevic, Z., Biely, M. and Schiper, A. [2013]. Bounded delay in byzantine-tolerant state machine replication, pp. 61–70.

[89] Milosevic, Z., Hutle, M. and Schiper, A. [2014]. Tolerating permanent and transient value faults, *Distrib. Comput.* **27**(1): 55–77.
URL: https://doi.org/10.1007/s00446-013-0199-7

[90] Moraru, I., Andersen, D. G. and Kaminsky, M. [2013]. There is more consensus in egalitarian parliaments, *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 358–372.

[91] Nakamoto, S. [2009]. Bitcoin: A peer-to-peer electronic cash system, *Cryptography Mailing list at https://metzdowd.com* .

[92] Naor, M. and Yogev, E. [2013]. Sliding bloom filters, *Algorithms and Computation*, Springer Berlin Heidelberg, pp. 513–523.

[93] Neves, N., Veríssimo, P., Gr, C., Correia, M., Ferreira, N. and Ver, N. [2004]. How to tolerate half less one byzantine nodes in practical distributed systems.

[94] Ongaro, D. and Ousterhout, J. [2014]. In search of an understandable consensus algorithm, *2014 USENIX Annual Technical Conference*, USENIX ATC 14, USENIX Association, pp. 305–319.

[95] Oppen, D. C. and Dalal, Y. K. [1983]. The clearinghouse: a decentralized agent for locating named objects in a distributed environment, *ACM Transactions on Information Systems (TOIS)* **1**(3): 230–253.

[96] Pass, R. and Shi, E. [2018]. *Thunderella: Blockchains with Optimistic Instant Confirmation*, pp. 3–33.

[97] Porto, D., Leitão, J. a., Li, C., Clement, A., Kate, A., Junqueira, F. and Rodrigues, R. [2015]. Visigoth fault tolerance, *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, Association for Computing Machinery, New York, NY, USA.
URL: https://doi.org/10.1145/2741948.2741979

[98] Ramasamy, H. V. and Cachin, C. [2006]. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast, Cryptology ePrint Archive, Paper 2006/082. https://eprint.iacr.org/2006/082.
URL: https://eprint.iacr.org/2006/082

[99] Schneider, F. B. [1990]. Implementing fault-tolerant services using the state machine approach: a tutorial, *ACM Computing Surveys* **22**(4): 299–319.

[100] *SGX* [n.d.]. https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html.

[101] Spiegelman, A., Giridharan, N., Sonnino, A. and Kokoris-Kogias, L. [2022a]. Bullshark: Dag bft protocols made practical, *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, Association for Computing Machinery, New York, NY, USA, pp. 2705–2718.
URL: https://doi.org/10.1145/3548606.3559361

[102] Spiegelman, A., Giridharan, N., Sonnino, A. and Kokoris-Kogias, L. [2022b]. Bullshark: The partially synchronous version, *ArXiv* **abs/2209.05633**.
URL: https://api.semanticscholar.org/CorpusID:252211739

[103] Stathakopoulou, C., David, T. and Vukolic, M. [2019]. Mir-bft: High-throughput BFT for blockchains, *CoRR* **abs/1906.05552**.
URL: http://arxiv.org/abs/1906.05552

[104] Stathakopoulou, C., Pavlovic, M. and Vukolić, M. [2022]. State machine replication scalability made simple, *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, Association for Computing Machinery, New York, NY, USA, p. 17–33.
URL: https://doi.org/10.1145/3492321.3519579

[105] Stathakopoulou, C., Wei, M. Y. C., Yin, M., Zhang, H. and Malkhi, D. [2023]. Bbca-ledger: High throughput consensus meets low latency, *ArXiv* **abs/2306.14757**.
URL: https://api.semanticscholar.org/CorpusID:259252419

[106] Team, L. E. [2018]. Libra: The path forward, Online. [Accessed 2020-06-01].
URL: https://libra.org/en-US/blog/the-path-forward/

[107] Thambidurai, P. M. and Park, Y.-K. [1988]. Interactive consistency with multiple failure modes, *Proceedings [1988] Seventh Symposium on Reliable Distributed Systems* pp. 93–100.
URL: https://api.semanticscholar.org/CorpusID:26823960

[108] van Renesse, R., Birman, K., Dumitriu, D. and Vogels, W. [2002]. Scalable management and data mining using astrolabe*, *Peer-to-Peer Systems*, Springer Berlin Heidelberg, pp. 280–294.

[109] van Renesse, R., Minsky, Y. and Hayden, M. [1998]. A gossip-style failure detection service, *Middleware'98*, Springer London, pp. 55–70.

[110] Verissimo, P. and Casimiro, A. [2002]. The timely computing base model and architecture, *IEEE Transactions on Computers* **51**(8): 916–930.

[111] Veronese, G. S., Correia, M., Bessani, A. N., Lung, L. C. and Verissimo, P. [2013]. Efficient byzantine fault-tolerance, *IEEE Transactions on Computers* **62**(1): 16–30.

[112] Veríssimo, P. [2003]. Uncertainty and predictability: Can they be reconciled?, Vol. 2584, pp. 108–113.

[113] Vogels, W., van Renesse, R. and Birman, K. [2003]. The power of epidemics, *ACM SIGCOMM Computer Communication Review* **33**(1): 131–135.

[114] Voulgaris, S. and van Steen, M. [2013]. Vicinity: A pinch of randomness brings out the structure, *Middleware 2013*, Springer Berlin Heidelberg, pp. 21–40.

[115] Wood, D. D. [2014]. Ethereum: A secure decentralised generalised transaction ledger.
URL: `https://api.semanticscholar.org/CorpusID:4836820`

[116] Yandamuri, S., Abraham, I., Nayak, K. and Reiter, M. K. [2021]. Communication-efficient bft protocols using small trusted hardware to tolerate minority corruption, Cryptology ePrint Archive, Paper 2021/184. `https://eprint.iacr.org/2021/184`.
URL: `https://eprint.iacr.org/2021/184`

[117] Yang, L., Park, S. J., Alizadeh, M., Kannan, S. and Tse, D. [2022]. DispersedLedger: High-Throughput byzantine consensus on variable bandwidth networks, *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, USENIX Association, Renton, WA, pp. 493–512.
URL: `https://www.usenix.org/conference/nsdi22/presentation/yang`

[118] Yin, M., Malkhi, D., Reiter, M. K., Golan Gueta, G. and Abraham, I. [2018]. Hotstuff: BFT consensus in the lens of blockchain.
URL: `https://arxiv.org/abs/1803.05069v6`

[119] Yin, M., Malkhi, D., Reiter, M. K., Gueta, G. G. and Abraham, I. [2019]. HotStuff: BFT consensus with linearity and responsiveness, *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC'19, pp. 347–356.