EPFL

Thesis n° 10 525

Robust and High-Performance Wide-Area Consensus Protocols

Presented on 15th August 2025

School of Computer and Communication Sciences
Decentralized and Distributed Systems Lab
Doctoral program in Computer and Communication Sciences

for the award of the degree of Docteur ès Sciences (PhD)

by

Pasindu Nivanthaka TENNAGE

Accepted on the jury's recommendation

Prof. P. Thiran, jury president
Prof. B. A. Ford, Dr E. Kokoris Kogias, thesis directors
Prof. N. Crooks, examiner
Prof. M. Kogias, examiner
Prof. C. Koch, examiner

École
polytechnique
fédérale
de Lausanne

2025

To my parents and sister...

Acknowledgements

First, I would like to express my sincere gratitude to my advisor, Prof. Bryan Ford. His guidance has been instrumental in my growth as a researcher. Bryan taught me the principles of systems research, how to design the right experiments, and how to present ideas convincingly. Most importantly, he encouraged me to always consider the broader impact of my work. Over the past 5.5 years, we have had many valuable discussions, and I am deeply grateful for all the support and guidance Bryan has provided throughout this time.

I would also like to sincerely thank my co-advisor, Dr. Lefteris Kokoris-Kogias, for his firm support in many ways. Lefteris consistently guided me on how to produce high-quality research and was always willing to listen to my concerns, both academic and personal. Throughout the past 5.5 years, we had numerous discussions, and I am truly grateful for his guidance, advice, and patience.

I am also deeply grateful to all the members of DEDIS for our insightful discussions and their support. In particular, I would like to thank Simone Colombo, Shailesh Mishra, and Louis-Henry for their valuable input and collaboration.

Finally, I would like to extend my heartfelt thanks to Raharisolo Fara and Sandra Hünsch for significantly easing my life by assisting with various tasks, especially administrative matters.

Abstract

Deploying consensus protocols in the wide-area is challenging due to unpredictable and adversarial nature of wide-area networks. This thesis explores five critical challenges that affect the operation of consensus protocols in the wide-area networks; (1) performance vulnerability of leader-based protocols to leader-targeted attacks, (2) losing liveness under adversarial network conditions, (3) throughput bottlenecks caused by leader overload, (4) recovery time versus liveness trade-off caused by sub-optimal manually configured timeouts, and (5) high commit latency in DAG-based asynchronous byzantine fault tolerant protocols.

To address these five challenges, this thesis proposes five novel protocols spread across four chapters. We first propose **Baxos**, a novel extension of the Paxos protocol, which replaces the traditional leader election in Multi-Paxos with a novel Random Exponential Backoff scheme. Baxos enables all nodes to propose, while handling contention using random exponential backoff, and enhances resilience against leader-targeted attacks. Second, we propose **SADL-RACS**, a novel modular state machine replication protocol. SADL-RACS overcomes the challenges of network asynchrony and leader performance bottleneck. **RACS** is a novel randomized crash fault-tolerant consensus protocol that guarantees liveness under network asynchrony. **SADL** eliminates the leader bottleneck by decoupling request dissemination from the critical path, distributing the load evenly across participating nodes. Third, we present **QuePaxa**, a novel randomized consensus algorithm that replaces manually configured timeouts in leader-based protocols with adaptive hedging delays. QuePaxa reduces recovery time during replica crashes or intermittent slowdowns while improving liveness under adversarial network conditions. Finally, we present **Mahi-Mahi**, a novel asynchronous DAG-based BFT protocol that achieves high throughput with sub-second commit latency. Unlike the certified DAG-based protocols, Mahi-Mahi employs an uncertified DAG to commit blocks, significantly reducing the number of network hops required per commitment.

Through protocol design, security proofs, prototype implementation, and extensive evaluation, this thesis demonstrates that each proposed solution delivers substantial improvements in performance and robustness. First, we show that Baxos improves throughput by up to 128% under leader-targeted attacks compared to Multi-Paxos and Raft. Second, we show that SADL-RACS ensures liveness under adversarial network conditions by providing a throughput of 196,000 requests per second. Third, we demonstrate that QuePaxa significantly outperforms the existing leader-based

protocols under adversarial network conditions, achieving at least 75,000 requests per second with a median latency of 380ms in wide-area deployments, while maintaining low recovery time, despite the choice of timeout value. Finally, Mahi-Mahi achieves over 100,000 transactions per second with sub-second average latency in wide-area deployments, setting a new milestone in asynchronous BFT protocol performance. These contributions transform high-performance wide-area consensus from a theoretical concept into a practical reality.

Key words: consensus, crash fault, state machine replication, byzantine, adversarial, networks, asynchrony.

Résumé

Le déploiement de protocoles de consensus en réseau étendu est difficile en raison de la nature imprédictible et hostile de ces réseaux. Cette thèse explore cinq défis critiques qui affectent le fonctionnement des protocoles de consensus en réseau étendu ; (1) la vulnérabilité des protocoles basés sur un leader aux attaques ciblant le leader, (2) la perte de vivacité dans des conditions de réseau hostiles, (3) la perte de débit causée par la surcharge du leader, (4) le compromis entre le temps de récupération et la vivacité causé par des délais d'attente configurés manuellement qui s'avèrent sous-optimaux, (5) la latence élevée de validation pour les protocoles tolérants aux pannes byzantines asynchrones basés sur un graphe acyclique dirigé (DAG).

Dans le but de résoudre ces défis, cette thèse propose cinq nouveaux protocoles répartis sur quatre chapitres. Nous proposons d'abord **Baxos**, une nouvelle extension du protocole Paxos qui remplace l'élection traditionnelle de leader dans Multi-Paxos par un nouveau schéma de recul exponentiel aléatoire. Baxos permet à tous les nœuds de proposer tout en gérant la contention à l'aide d'un schéma de recul exponentiel aléatoire et en améliorant la résistance face aux attaques visant le leader. Deuxièmement, nous proposons **SADL-RACS**, un nouveau protocole modulaire de réplication de machine à état. SADL-RACS permet de s'affranchir des problèmes d'asynchronie du réseau et de la dégradation des performances en cas de surcharge du leader. **RACS** est un nouveau protocole de consensus randomisé tolérant aux pannes et garantissant la vivacité dans un réseau asynchrone. **SADL** élimine le goulot d'étranglement causé par le leader en découplant la dissémination de requêtes du chemin critique, distribuant ainsi la charge de façon uniforme parmi les nœuds participants. Troisièmement, nous présentons **QuePaxa**, un nouveau protocole de consensus randomisé qui remplace la configuration manuelle des délais par des délais d'attente adaptatifs redondants. QuePaxa réduit le temps de récupération en cas de perte de réplica ou de ralentissement intermittent, tout en améliorant la vivacité dans les conditions de réseaux hostiles. Enfin, nous présentons **Mahi-Mahi**, un nouveau protocole asynchrone résistant aux adversaires byzantins basé sur un graphe direct acyclique. Mahi-Mahi permet d'obtenir de hauts débits avec des latences de validation inférieures à la seconde. Contrairement aux protocoles basés sur DAG certifiés, Mahi-Mahi utilise un DAG non certifié pour valider les blocs, réduisant considérablement le nombre de sauts de réseau requis par validation.

À travers la conception de nouveaux protocoles, les preuves de sécurité, l'implémentation de prototypes et l'évaluation exhaustive, cette thèse démontre que chacune des solutions proposées délivre des améliorations substantielles en termes de performance et de robustesse. Premièrement,

nous montrons que BAXOS améliore le débit par un facteur allant jusqu'à 128% par rapport à Multi-Paxos et Raft dans des conditions d'attaques ciblées sur le leader. Deuxièmement, nous montrons que SADL-RACS garantit la vivacité malgré un réseau hostile en offrant un débit de 196 000 requêtes par seconde. Troisièmement, nous démontrons que QuePaxa permet d'obtenir des performances significativement plus élevées que les protocoles existants à base de leader dans le cadre de réseaux hostiles, atteignant plus de 75 000 requêtes par seconde avec une latence moyenne de 380 ms sur un réseau étendu tout en maintenant un temps de récupération faible peu importe le choix de timeout. Enfin, Mahi-Mahi atteint les 100 000 transactions par seconde avec une latence moyenne de moins d'une seconde sur réseau étendu, définissant ainsi une nouvelle référence en termes de performance pour les protocoles BFT asynchrone. Combinées, ces contributions transforment le concept théorique de consensus haute performance en réseau étendu en une réalité pratique.

Mots clés : consensus, résistance aux pannes, réplication de machine à état, byzantin, hostile, réseaux, asynchronicité.

Contents

List of Figures

List of Tables

1 Introduction

Consensus protocols [1], [2] form the foundation of reliable coordination in distributed systems, enabling a set of replicas to agree on a single history of operations, despite replica failures. Consensus is central to distributed applications such as databases [3], [4], distributed filesystems [5]–[7], and social networks [8], [9].

Building on consensus, State Machine Replication (SMR) allows replicas to maintain a strongly consistent state by executing the agreed-upon commands in the same sequential order. SMR is a critical technique for ensuring fault tolerance and high availability [10]–[12].

Despite extensive research, the adoption of consensus protocols and state machine replication in wide-area networks presents significant challenges. Existing protocols often struggle to deliver high performance in geo-replicated settings, facing reliability issues induced by wide-area network conditions such as asymmetric connectivity, high jitter, and unreliable links. Although consensus research has evolved over more than four decades, these practical challenges have limited its widespread adoption.

This thesis addresses the challenges of deploying consensus algorithms in wide-area networks, focusing on both performance and robustness. Under performance challenges, this thesis examines the limitations of leader-based consensus protocols such as Multi-Paxos [13] and Raft [14], as well as geo-replicated blockchain protocols like Tusk [15] and Bullshark [16]. For robustness, this thesis investigates the susceptibility of consensus protocols to adverse network conditions, including high jitter, high latency, and targeted denial-of-service attacks [17].

1.1 Challenges of wide-area consensus

This thesis identifies five key challenges affecting the performance and robustness of wide-area consensus protocols;

- Many widely deployed consensus protocols [13], [14], [18], are leader-based, exposing them to leader-targeted attacks that can severely degrade the performance.
- Widely deployed protocols generally provide liveness assuming a partially synchronous network model, hence compromising liveness under adversarial network conditions which are prevalent in the wide-area setting.
- Leader-based protocols often fail to achieve optimal performance due to the high load imposed on the leader replica, resulting in throughput that falls short of theoretical expectations.
- The performance of existing protocols is highly sensitive to the selection of the timeout parameter, leading to a trade-off between recovery time and liveness depending on the chosen timeout value.
- DAG based Byzantine Fault Tolerant protocols often exhibit high latency, with wide-area commit latencies frequently exceeding several seconds, making them unsuitable for applications that require low-latency operations.

In the following subsections, we delve into these five challenges that impact the performance and robustness of wide-area consensus protocols.

1.1.1 Leader-targeted attacks

For performance reasons, most deployed consensus protocols use a leader which handles both client requests and inter-replica messages [14], [19], [20]. However, under targeted leader-based attacks, leader-based design becomes problematic [21], [22] – when the leader is attacked by a distributed denial-of-service (DDoS) attack, leader-based approaches fail to deliver good performance due to leader timeouts and subsequent leader election mechanisms which impact the overall system availability and performance. In the worst case, a service using the consensus protocol can even freeze completely [23].

Thus, our first research question is: *Can the impact of leader-targeted attacks on performance in leader-based consensus protocols be mitigated or minimized?*

1.1.2 The asynchrony challenge

Widely-used leader-based [13], [14], [20] and multi-leader protocols [24], [25], rely on a partially synchronous network model to ensure liveness. Specifically, leader-based protocols complete

view changes only under partial synchrony, while multi-leader protocols require partial synchrony to resolve conflicting concurrent proposals and node failures [24], [25].

In wide-area networks, network conditions can frequently become adversarial due to transient slowdowns and network misconfigurations [23]. Under such conditions, where network delays become highly unpredictable, both leader-based and multi-leader protocols fail to maintain liveness.

Theoretical work has proposed leaderless randomized protocols [26], [27] that guarantee liveness even under adversarial networks. However, their normal-case performance suffers due to quadratic message complexity, rendering them impractical for real-world deployments.

Therefore, our second research question is:

Can we design practical consensus protocols that under synchronous networks, perform comparable to Multi-Paxos and Raft, while guaranteeing liveness under asynchronous network conditions?

1.1.3 The leader performance bottleneck challenge

The throughput of existing leader-based protocols [13], [14], is limited by the leader replica's network bandwidth and computational capacity. As the number of replicas increases beyond 3–5, these protocols experience a significant drop in throughput due to the leader's resource constraints. Consequently, leader-based protocols inherently face a trade-off between throughput and fault-tolerance: a low replication factor allows for higher throughput but offers limited fault tolerance, while a high replication factor improves fault tolerance at the cost of reduced throughput.

Thus, our third research question is: *Can we design consensus protocols that simultaneously achieve both high throughput and high fault-tolerance?*

1.1.4 The tyranny of timeout

Mainstream SMR protocols heavily rely on timeouts, resulting in three key challenges collectively referred to as the *tyranny of timeouts*. First, existing widely-deployed protocols rely on timeout for liveness: (1) leader-based and multi-leader protocols use timeout configured view changes to provide liveness and (2) hybrid protocols that switch between leader-based and randomized protocol stacks, involve a timeout to switch between the two modes [28]. Second, simultaneous leaders can cause destructive interference, and view changes come with a high cost. To mitigate these risks, timeouts must be conservatively large, which can hinder performance during leader failures and subsequent recovery. Third, timeouts require meticulous manual configuration. Misconfigurations can lead to degraded performance or even complete system failure, imposing a significant operational burden.

Therefore, our fourth research question is: *Can consensus protocols be designed to escape the tyranny of timeouts?*

1.1.5 High latency cost of DAG based randomized BFT protocols

BFT protocols address the asynchronous network challenge by leveraging randomness [15], [28], [29]. State-of-the-art high-performance BFT protocols utilize a Directed Acyclic Graph (DAG) architecture [15], [27], which improves throughput by committing a larger history of blocks compared to linear chain protocols. However, existing DAG based randomized BFT protocols often suffer from high latency, typically exceeding 3 seconds in the wide-area network deployments.

A key factor contributing to this high latency lies in the architectural design of DAG-based protocols. These protocols typically employ a certified architecture [15], [27], which necessitates running a reliable broadcast protocol in addition to the consensus mechanism. Uncertified DAG based protocols [29], [30] address this issue by enabling commitment over a non-certified DAG, however, supports only partially synchronous network settings [30] or incur high number of rounds per consensus decision [29], thus making them impractical for latency-sensitive applications.

Thus, our fifth and final research question is: *Can we design a practical randomized DAG-based BFT protocol that achieves low latency in the wide-area?*

1.2 Road Map of Thesis Contributions

To address the five challenges outlined above, this thesis proposes five solutions, which are presented across four chapters. Each solution contributes to the overall goal of improving the performance and resilience of wide-area consensus protocols.

1.2.1 Eliminating the impact of leader-targeted attacks using Baxos

Chapter 3 addresses the challenges inherent in leader-based protocols (challenge 1.1.1). Specifically, we investigate the feasibility of replacing the leader election mechanism in Multi-Paxos with Random Exponential Backoff (REB) — a simpler and more resilient approach that requires minimal modifications to the two-phase Paxos algorithm [13].

This thesis introduces **Baxos**, a novel consensus protocol that replaces leader election in Multi-Paxos with a REB scheme. By eliminating leader election, REB allows each replica to propose commands independently, mitigating the impact of leader-targeted attacks. Our design addresses key challenges associated with REB, including scalability and adaptability to fluctuating wide-area latencies.

To demonstrate its effectiveness, an extensive evaluation of Baxos is conducted using real-world deployments on Amazon EC2 across a wide-area network. The evaluation includes a combination of micro benchmarks and the YCSB-A workload on Redis [31]. Our results show that Baxos is significantly more robust against liveness and performance downgrade attacks compared to Multi-Paxos and Raft. Specifically, under leader-targeted performance downgrade attacks, Baxos achieves up to a 128% improvement in throughput compared to Multi-Paxos and Raft.

1.2.2 Achieving liveness under asynchronous network conditions using RACS

Chapter 4 presents RACS, a novel consensus protocol designed to address the network asynchrony challenge (challenge 1.1.2).

RACS is a novel crash fault-tolerant consensus algorithm. RACS operates in two modes: synchronous and asynchronous, ensuring liveness in all conditions. In synchronous mode, RACS minimizes communication costs to $O(n)$, while in adversarial scenarios, it matches the theoretical lower bound of quadratic message complexity, providing robust performance under varying network conditions.

The evaluation on Amazon EC2 shows that RACS significantly improves robustness, sustaining over 28,000 requests per second under adversarial network conditions, while existing leader-based protocols, Multi-Paxos and Raft, lose liveness entirely.

1.2.3 Avoiding leader-bottleneck using SADL

To tackle the leader performance bottleneck challenge 1.1.3, the second part of chapter 4 proposes SADL, a consensus-agnostic asynchronous dissemination layer. SADL decouples client command dissemination from the critical path of consensus, distributing the bandwidth overhead evenly across all replicas. Decoupling command dissemination from the critical path of consensus allows the consensus protocol to scale to a higher number of replicas without sacrificing the throughput.

The evaluation conducted in Amazon EC2 demonstrates that SADL-RACS–the version of RACS that runs on top of SADL– achieves significant performance improvements, maintaining over 196,000 requests per second even under adversarial conditions, while also supporting scalability beyond the limits of traditional leader-based protocols.

1.2.4 Avoiding the tyranny of timeouts using Quepaxa

Chapter 5 addresses the tyranny of timeouts challenge (1.1.4), focusing on overcoming the limitations imposed by timeout-based design in leader-driven consensus protocols. QuePaxa introduces a novel randomized asynchronous consensus core designed to withstand adverse conditions, such as DdoS attacks, while maintaining the normal-case efficiency of protocols like Multi-Paxos and Raft through a one-round-trip fast path, under synchronous executions.

QuePaxa allows simultaneous proposers without destructive interference and replaces conservative timeouts with short hedging delays to limit redundant effort. This approach enables rapid recovery following leader failures, avoiding the costly view changes typically triggered by false timeouts. By framing leader selection and hedging delay as a multi-armed bandit optimization problem, QuePaxa dynamically adapts to changing network and node conditions and can preemptively switch to a more suitable leader, even before the current leader fails.

Experimental evaluations of QuePaxa demonstrate its effectiveness. In normal-case scenarios, QuePaxa achieves throughput of 584k cmd/sec in LAN and 250k cmd/sec in WAN, comparable to Multi-Paxos. Under challenging conditions such as DoS attacks, network misconfigurations, or slow leader performance—scenarios that significantly degrade existing protocols — QuePaxa maintains liveness, achieving a median latency of under 380ms in WAN environments. These results highlight QuePaxa's ability to combine high performance with robustness in a wide range of network conditions.

1.2.5 Low latency DAG based consensus using Mahi-Mahi

Chapter 6 proposes Mahi-Mahi, which addresses the high latency challenge of DAG based BFT protocols (1.1.5). Mahi-Mahi is the first asynchronous BFT consensus protocol to achieve sub-second latency in wide-area networks while processing over 100,000 transactions per second. This exceptional performance is achieved by utilizing two techniques – multi-leader and pipelining on top of the the uncertified DAG structure. This approach significantly reduces the number of messages required for committing transactions.

Mahi-Mahi introduces a novel commit rule that enables multiple blocks to be committed in each DAG round, ensuring liveness even in the presence of an asynchronous adversary. The protocol can be parametrized to either commit within 5 message delays, maximizing the probability of commitment under a continuously active adversary, or within 4 message delays, reducing latency under more moderate adversarial conditions.

This thesis evaluates Mahi-Mahi's performance in a geo-replicated setup and compares it against state-of-the-art asynchronous consensus protocols. Our evaluation highlights Mahi-Mahi's ability to consistently achieve sub-second latency, while reaching 100,000 requests per second throughput, making it a practical and efficient protocol for high-performance blockchains.

1.3 Limitations and Scope of the Thesis

We provide several novel solutions to address the challenges inherent in wide-area consensus protocols. However, this thesis does not claim to break any theoretical bounds on consensus. **Specifically, we do not improve the theoretical message complexity or round complexity of any protocol under any network model.** Instead, our focus is on designing and building practical, pragmatic consensus protocols that can be readily deployed in wide-area environments.

While this thesis proposes several key contributions toward achieving high-performance and robust wide-area consensus protocols, we acknowledge the existence of other mechanisms—many of which are orthogonal to our goals—that enhance the performance and robustness of wide-area consensus. Notably, techniques such as flexible quorums [32], which reduce quorum size in the common case, sharding-based approaches [33], which partition the object space across multiple parallel SMR instances, and broadcast tree-based methods [34], which improve message propagation using a tree-structured broadcast, are valuable contributions in this domain, however, fall outside the scope of this thesis.

While our focus is on wide-area consensus protocols, we also recognize the importance of innovations in data center consensus, where network conditions are typically more stable but performance expectations are generally much higher. Protocols such as Ring-Paxos [35], No-Paxos [36], and HovercRaft [37] leverage advancements in data center networks to enhance performance, offering valuable insights for improving consensus in such environments. These contributions are outside the scope of this thesis.

1.4 Thesis statement

Existing consensus protocols face a tradeoff between high performance and robustness. Protocols designed for high performance often rely on leader-based designs, making them vulnerable to asynchronous network conditions. In contrast, protocols built for robustness typically use randomization to ensure liveness under asynchrony but suffer from significant performance overhead. This thesis addresses this tradeoff by proposing four novel consensus protocols—Baxos, SADL-RACS, QuePaxa, and Mahi-Mahi—that achieve both high performance and robustness. Through protocol design, system implementation, and evaluation on Amazon EC2, we show that our new protocols sustain high performance while remaining robust under adverse network conditions. Hence, both theoretically and empirically, this thesis demonstrates that the tradeoff between high performance and robustness is not fundamental and can be overcome.

1.5 Summary and Contributions

In summary, this thesis makes the following contributions;

- To avoid the impact of leader-targeted attacks on the performance of leader-based protocols, this thesis proposes the design and evaluation of **Baxos** – a novel leaderless variant of Paxos that uses REB as a replacement for leader-election.
- To avoid the liveness challenges inherent in adversarial networks, and to avoid the leader performance bottleneck, this thesis proposes **SADL-RACS**, a modular, asynchronous wide-area SMR system.
- To avoid the tyranny of timeouts, this thesis proposes the design and evaluation of **QuePaxa**, a

novel randomized consensus protocol.

- To achieve sub-second latency in the wide-area DAG based blockchains, this thesis proposes **Mahi-Mahi**, a novel uncertified DAG based consensus protocol.

The contributions of this thesis collectively make wide-area consensus a more practical tool rather than a purely theoretical concept. **This thesis emphasizes that prioritizing only one of performance or robustness alone is no longer sufficient**; **practical consensus protocols must be co-designed to ensure both**. Although several open research questions remain in the realm of wide-area consensus, this work addresses some of the most pressing challenges. These contributions are expected to pave the way for future research in this field.

2 Background

This chapter presents the background relevant to the subsequent chapters of this thesis. We begin by discussing the fundamentals of consensus and SMR, followed by a discussion of the different network models and the FLP impossibility result. We conclude the chapter with a brief summary of permissioned and permissionless consensus protocols.

2.1 Consensus

Consensus is a fundamental abstraction in distributed systems, enabling a group of nodes to reach an agreement on a single value despite potential failures in the network and nodes. The problem arises in systems where multiple nodes operate independently but need to maintain a strongly consistent state or make a collective decision, such as in distributed databases [3], [4], [38], distributed filesystems [5]–[7], and social networks [8], [9].

A consensus protocol allows each node in the system to propose a value, which could represent a transaction or a command. The protocol ensures that the nodes agree on a single value, despite the presence of multiple proposals.

A correct consensus algorithm satisfies four properties [2]:

- Validity: a decided value should be previously proposed by a process.
- Agreement: no two correct processes decide differently.
- Termination: every correct process eventually decides some value.
- Integrity: no process decides twice.

2.2 State Machine Replication

State Machine Replication (SMR) is a technique in distributed computing, enabling fault-tolerant and consistent operation of services across distributed replicas. SMR involves maintaining identical copies of a state machine on multiple nodes and ensuring that all replicas process the same sequence of inputs in the same order. This deterministic behavior ensures consistency, as all replicas transition through the same states and produce identical outputs. SMR employs consensus protocols to achieve agreement on the order of commands.

A correct SMR algorithm satisfies two properties:

- Safety: no two replicas commit different client commands for the same log position.
- Liveness: each client command is eventually committed.

We assume that each client command will be repeatedly proposed by replicas until it is committed.

2.3 Fault Modes

Consensus protocols can be broadly categorized into two types based on the fault model they address: (1) Crash Fault Tolerant (CFT) and (2) Byzantine Fault Tolerant (BFT) consensus.

CFT consensus focuses on handling crash faults, where nodes cease functioning but neither act maliciously nor omit messages. Protocols such as Paxos [13] and Raft [14] are designed to guarantee safety and liveness despite such crash faults. To tolerate up to f crash faults, a CFT consensus protocol requires at least $n = 2f + 1$ number of replicas. *The first three chapters of this thesis (chapters 3,4,5) focus on the CFT model.*

BFT consensus addresses a more challenging fault model, where nodes can behave arbitrarily, including exhibiting behaviors such as equivocation and omission[2]. BFT protocols, such as PBFT [39], Hotstuff [18], and Dag-rider [27] provide safety guarantees even in the presence of Byzantine faults, making them suitable for applications requiring higher security, such as blockchain systems. To tolerate up to f Byzantine faults, a BFT consensus protocol requires at least $n = 3f + 1$ nodes. While BFT protocols offer stronger fault tolerance, they typically involve higher communication overhead and protocol complexity compared to CFT protocols. *In this thesis, the chapter 6 focuses on the BFT model.*

There are other standard failure models, such as crash-recovery [2], where a node can rejoin the consensus replica set after crashing; eavesdropping faults [2], where a malicious process leaks information obtained during the consensus protocol to an external entity, and rational faults [40]–[42], where a replica's actions depend on economic incentives. However, for most practical applications, crash faults and Byzantine faults adequately capture the predominant failure modes. Therefore, this thesis focuses exclusively on CFT and BFT models.

2.4 Network Models

The liveness of consensus protocols is often discussed with respect to different network models. A widely accepted classification [43] uses the message transmission delay Δ to define the characteristics of each network model. In this section, we describe the three most commonly cited network models.

Let Δ be the message transmission delay and GST be the global stabilization time.

- An execution of a protocol is considered *synchronous* if for every run R of the protocol, each message sent from a correct replica p_i is delivered by correct replica p_j within a bounded Δ.
- An execution of a protocol is considered *asynchronous* if for any run R of the protocol, there exists no time bound Δ for message delivery.
- An execution of a protocol is considered *partially synchronous*, if at least one of the following conditions hold;
  - For every run R, there is a time GST such that Δ holds in $[GST, \infty)$ (i.e. Δ holds eventually for some fixed Δ).
  - For every run R, there is a Δ that holds in $[1, \infty)$ (i.e. Δ is unknown) [43].

In this thesis, Baxos (Chapter 3) assumes a partially synchronous network model to ensure liveness, whereas SADL-RACS, QuePaxa, and Mahi-Mahi (Chapters 4,5,6) provide liveness under an asynchronous network model.

2.5 Circumventing FLP Impossibility

The FLP impossibility result [44], established by Fischer, Lynch, and Paterson, is a fundamental theorem in distributed computing and states that no deterministic protocol can solve consensus in an asynchronous network with even a single node failure. FLP result highlights the inherent limitations of achieving consensus in distributed environments where there is no bound on message delivery time.

Given the FLP impossibility result, there are two common approaches to circumvent it:

- Assume partial synchrony: In this approach, the protocol assumes partial synchrony to ensure liveness. Examples of such protocols include Multi-Paxos [13], Raft [14], and HotStuff [18]. These protocols ensure **liveness** only when the network eventually becomes synchronous.

  In this thesis, Baxos (chapter 3) assumes partial synchrony to ensure liveness.

- Use Randomization: Randomized protocols leverage randomization to prioritize and break ties between competing proposals. They allow multiple proposers and progress in a stepwise fashion, advancing to the next logical step only when a majority of nodes have completed the current step [26], [27].

  In this thesis, RACS (chapter 4), QuePaxa (chapter 5), and Mahi-Mahi (chapter 6) employ randomization to alleviate the FLP impossibility and ensure liveness even under an asynchronous network.

2.6 Permissioned versus Permissionless Consensus

Permissioned and permissionless consensus models address different membership assumptions. **Permissioned consensus** protocols operate within a closed network where membership is agreed upon beforehand. Protocols such as PBFT [39], Hotstuff [18], Bullshark [16] achieve high efficiency and low overhead by leveraging this pre-established membership. Furthermore, permissioned consensus protocols allow the replica set to be reconfigured through explicit membership protocols, typically executed atop the SMR layer [14]. *In this thesis, we assume a permissioned setting for all four chapters.*

Permissionless consensus protocols enable decentralized networks to reach consensus without requiring prior authorization for participation. Permissionless protocols allow any node to join the network and contribute to the consensus process. Common approaches to permissionless consensus include Proof of Work, Proof of Stake, and Proof of History.

Proof of Work (PoW) requires participants, known as miners, to solve cryptographic puzzles, independently from each other. The block from the first node which solves the cryptographic puzzle is appended to the ledger as the next block. This mechanism secures the distributed ledger by making it computationally expensive to alter the blockchain's history, hence providing eventual safety [45]–[47]. Proof of Stake (PoS) assigns the right to create new blocks based on the amount of stake a participant holds and is willing to lock up as collateral [48]–[50]. This approach reduces energy consumption compared to PoW. Proof of History (PoH) introduces a verifiable delay function to establish a historical record that proves events have occurred in a specific sequence [51].

3 Baxos: Backing off for robust consensus

Leader-based consensus algorithms are vulnerable to liveness and performance downgrade attacks [22]. We explore the possibility of replacing leader election in Multi-Paxos with *random exponential backoff (REB)*, a simpler approach that requires minimum modifications to the two-phase Paxos [13] and achieves better resiliency under attacks.

We propose Baxos, a new resilient consensus protocol that leverages a random exponential backoff scheme as a replacement for leader election in consensus algorithms. Our backoff scheme addresses the common challenges of random exponential backoff such as scalability and robustness to changing wide-area latency. We extensively evaluate Baxos to illustrate its performance and robustness against liveness and performance downgrade attacks using an implementation running on Amazon EC2 in a wide-area network and a combination of a micro benchmark and YCSB-A workload on Redis. Our results show that Baxos offers more robustness to liveness and performance downgrade attacks than leader-based consensus protocols. Baxos outperforms Multi-Paxos and Raft up to 128% in throughput under liveness and performance downgrade attacks under worst case contention scenarios where each replica proposes requests concurrently while incurring up to 32% reduction on the maximum throughput in the synchronous attack-free scenario.

The content of this chapter appears in a public preprint available at https://arxiv.org/pdf/2204.10934. This work was done in collaboration with Cristina Basescu, Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, and Bryan Ford.

3.1 Introduction

For performance reasons, most deployed consensus protocols use a leader which serves client requests and inter-replica messages [14], [19], [20]. In particular, the leader is tasked with handling contention and providing lock-free termination, which works well in synchronous and attack-free network settings. However, under more adversarial network conditions, this approach becomes problematic [21], [22]. When the network is volatile, e.g., changing link delays and bandwidth, leader-based approaches fail to deliver good performance due to leader timeouts and subsequent leader election mechanisms which impact the overall system availability. In the worst case, a service can even freeze completely, which is exactly what happened in a Cloudflare outage [23]. This downside becomes particularly problematic when a system is under a distributed denial of service (DDoS) attack. With DDoS attacks becoming more prevalent [17], leader-based consensus algorithms pose a significant risk to the availability of internet applications.

Previously proposed consensus algorithms that achieve lock-free termination without using a leader node include multi-leader-based protocols [25], [34], [52], [53], sharding based protocols [33], [38], [54], [55], protocols that exploit request dependencies [24], [56], [57] and asynchronous algorithms [26], [58]. The existing approaches fail to provide robustness against targeted leader-based attacks. Most multi-leader-based algorithms delegate message propagation to other replicas but still rely on a leader to order requests [34], [52], [53], remaining susceptible to attacks on the leader node. Algorithms that exploit request dependencies are vulnerable to DDoS attacks that issue concurrent dependent requests [24], [59]. DDoS attacks against the top level shards in sharding-based consensus algorithms [33], [38], [54], [55] can make the entire system unavailable. Finally, fully asynchronous algorithms are generally complex, rarely implemented, and usually do not perform as well as Multi-Paxos in practice.

We observe that even after two decades of leaderless consensus protocol research, the majority of the deployed consensus algorithms still use leader-based protocols such as Multi-Paxos or Raft [11], [60]. This situation has led us to investigate the minimal modification required to transform a consensus algorithm such as Paxos [13] to a consensus algorithm that is robust against liveness and performance downgrade attacks while preserving good performance in an attack-free scenario. In turn, we explore the possibility of utilizing random exponential backoff (REB) [61] in the context of consensus, due to its robustness, efficient contention handling, and power efficiency guarantees. As a result, we propose Baxos, a bare minimal modification of Paxos [13] that is robust and highly available under liveness and performance downgrade attacks.

Baxos employs the same two-phase protocol core as Paxos, but in contrast to Multi-Paxos, it uses REB instead of leader election to achieve lock-free termination. In Baxos, every node can propose values and, when concurrent proposals collide, they *back off* to avoid further collisions, an approach similar to CSMA in LANs [62]. Replacing leader election with random exponential backoff is not trivial, however, due to its potential side effects such as (1) the *capture effect*, where a single node can have an unfair share of a shared resource as well as (2) the impact on resilience to changing network delays, and (3) scalability. Baxos leverages a REB protocol that scales up to

nine replicas while remaining resilient to changing network delays and minimizing the capture effect.

Baxos is the first attempt to prototype REB-based Paxos and to systematically explore its properties. To evaluate the properties of Baxos, we compare Baxos against Multi-Paxos [13] and Raft [14] . We first analyze the performance of Baxos under *delayed view change attacks*, a class of targeted performance downgrade attacks in the wide-area, and show that Baxos, in such a situation, significantly outperforms Multi-Paxos and Raft by up to 128% in throughput. Then, we explore the performance overhead of Baxos under attack free synchronous network scenarios in the wide-area, and show that it achieves a throughput of 17,500 requests per second in contrast to the 28,000 requests per second saturation throughput of Multi-Paxos and Raft. Third, we analyze the uniformity of bandwidth utilization and show that Baxos achieves a more uniform resource utilization across a set of consensus replicas than Multi-Paxos and Raft. Finally we show that Baxos can scale up to nine nodes in the wide-area.

To summarize, this chapter makes the following contributions:

- We explore the use of REB as a replacement for the leader election in consensus algorithms.
- We design and systematically develop a consensus algorithm by combining Paxos and REB.
- We provide an experimental analysis of Baxos under both adversarial and normal-case network conditions.

3.2 Background

This section provides an overview of leader-based protocols and its performance vulnerabilities, as well as the random exponential backoff mechanism we use as a building block of Baxos.

3.2.1 Leader-Based Consensus

Multi-Paxos [13] and Raft [14] are the most widely deployed consensus algorithms [11], [60] that rely on partial synchrony to alleviate the FLP impossibility result. In Multi-Paxos, a replica runs the Prepare-Promise phase for a sequence of consensus instances in the leader election phase and becomes the leader. Then, each client request is committed in the Propose-Accept phase in a single round trip. Raft builds on top of view-stamp replication [63]. When the leader is stable, Raft achieves a single round trip time consensus. When the leader fails, Raft uses a leader election algorithm to elect a new leader. On a high level, both Multi-Paxos and Raft solve the consensus problem in a similar method, differing only in the way a new leader is elected [64].

3.2.2 Performance Vulnerabilities

Consensus protocols are often deployed across wide area networks using the (public) Internet infrastructure to achieve high availability through replication. Networks, however, can be impacted by different adverse network conditions, ranging from accidental (e.g., a network congestion can affect the communication to and from the current leader slowing down all nodes) to intentional (e.g., a carefully crafted DDoS attack can interfere with a consensus replica group).

DDoS is a relatively simple but powerful technique to attack Internet resources [17], preventing or limiting access to a targeted resource. In the context of consensus, an attacker can perform a DDoS attack by carefully analyzing the traffic using traffic analysis, and attacking the leader node to degrade the performance of the system by forcing the replicas to follow the slow execution paths such as view change [21].

We make use of the DDoS attack description of Spiegelman *et al.* [22] to represent DDoS attacks relevant to consensus. We will refer to an attack that affects a consensus protocol as a *delayed view change attack*. A delayed view change attack aims to degrade the performance of a consensus algorithm *while* maximizing the time it takes to elect a new leader by (1) saturating the resources of leader replica and (2) avoiding a view change for the maximum possible amount of time. Saturating the leader replica in a consensus system slows down the entire replica set. However, leader-based consensus algorithms are configured to trigger a view change to elect a new leader when the current one becomes unresponsive for a predefined time period. If the attacker targets the leader in a way that immediately triggers a view change, then the new leader will keep the system available, foiling the attack. Hence, the attacker has to consider the trade-off between the performance loss due to the attack and the frequency at which a new leader is elected. Delayed view change attack differs from regular leader failures such that in the regular leader failures the leader node is permanently made unavailable where as in the delayed view change attack the leader node is slowed down temporarily for a time duration that is less than view change time. While the effect of permanent leader failure is widely explored in the previous work [25], we found that the effect of delayed view change attack has not been explored in the previous work.

3.2.3 Random Exponential Backoff (REB)

REB is a mechanism that enables a set of nodes to consume a shared resource without relying on a centralized point of entry. REB emerged as a standard technique to access shared resources in Ethernet [65] and the DOCSIS cable network [66]. In Ethernet, when there are concurrent data transmissions in the shared data link medium, the nodes detect the collision and re-transmit the frame. To avoid further collisions, each node backs off a random amount of time, exponentially increasing the random timeout duration.

In contrast with its firm establishment in networking, REB has not been well studied in the context of consensus. Exponential timeouts have been used in consensus protocols but mainly as a method to *adjust* the leader timeouts. Multi-Paxos [13], [67] and Raft employ random

exponential timeouts for two reasons: (1) to increase the view change timeout upon each view change and (2) to avoid two replicas concurrently issuing a new view change. However, none of the previous work have explored REB as a leader *replacement* method, and to the best of our knowledge, our work is the first attempt to leverage and thoroughly evaluate REB as a primary method of contention handling in consensus.

Before deriving our own REB scheme, we first considered the existing REB algorithms used in the networking literature [65]. Binary REB is a fundamental algorithm in this space. In binary REB, each node backs off a random time out that increases exponentially upon each successive retry.

Binary REB algorithm has a major limitation: the *capture effect* [68]. Binary REB resets the variable *retries* (failed proposals) to zero upon a successful transmission by a node. However, this gives unfair advantage to the node which successfully transmitted its message, by having a low *contention set* w.r.t other nodes, thus enabling it to transmit subsequent messages, while the other nodes are backing off with a larger contention set. In networking literature this phenomenon is called the capture effect. This limitation of binary REB affects the fairness: only some replicas succeed in proposing values.

We also observed that existing REB protocols in the CSMA literature are designed with strict network synchrony assumptions. This assumption holds true in CSMA because all the nodes sharing a data link medium are tightly synchronized to the closest micro second. However, applying these algorithms to Baxos is not trivial due to changing wide-area network latency. Due to these limitations of existing REB schemes, we modified and adapted the binary REB scheme, and refer it as Baxos REB (see Section 3.3.3).

3.3 Design

In this section we first describe Baxos's system model followed by the algorithm itself. Afterwards we describe how REB integrates with Baxos, provide the consensus proofs, and finally discuss an optimization.

3.3.1 System Model

Let n denote the number of replicas and let f denote the fault tolerant threshold. We assume $n = 2f + 1$ and crash stop failures. For simplicity, we further assume that crashes are permanent although node recovery can be easily integrated into Baxos using standard recovery approaches like sync-on-disk for each operation [2], [24].

We assume perfect point-to-point links between each pair of nodes, i.e., messages sent to non-failed nodes are eventually delivered [2]. We also assume a partially-synchronous network as defined in Dwork *et al.* [43].

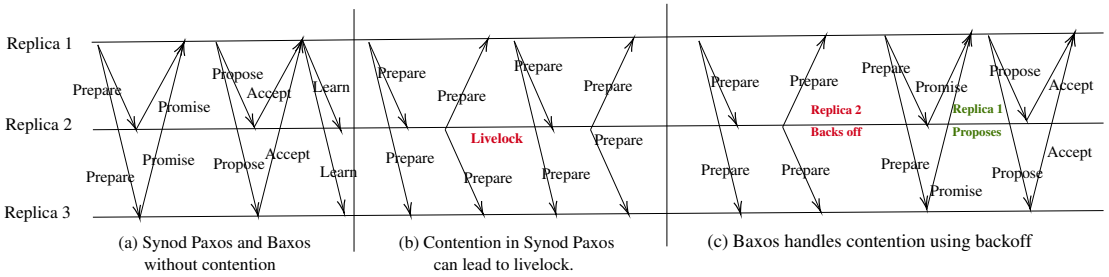


Figure 3.1: A space time diagram showing the message flow of Synod Paxos and Baxos with and without contention.

3.3.2 The Baxos Algorithm

At its core, Baxos uses the Synod Paxos protocol (*Synod Paxos*) [69], where each replica can propose values. However, Synod Paxos fails to achieve liveness if there are concurrent proposals for the same consensus instance. Baxos addresses this liveness issue by using REB: if there are concurrent requests for the same consensus instance, Baxos replicas *back off* for a random amount of time to prevent further collisions. This ensures that one proposer eventually succeeds in committing their value for the consensus instance within a few retries. We present the pseudo-code in Algorithm 1.

We use the term *try* to denote the concept of *Ballot* number in Synod Paxos, and the term *choice* to indicate a consensus instance. A sequence of *choice* elements make the replicated log. As in Paxos, each replica can take on the role of a *Acceptor*, *Proposer*, and *Learner* [13].

Single-choice Baxos consist of the following two phases, see Figure 3.1(a).

Prepare-Promise A node which receives a new command from the upper layer takes on the role of Proposer and initiates consensus by broadcasting a Prepare message to all Acceptors. The Prepare message contains a *proposed_try* number, which keeps track of the current *try* number. Acceptors send a Promise message to the Proposer, if they have not accepted any Prepare message with a higher or equal *try* number than the *proposed_try* received in the Prepare message. To inform the Proposer about any previously accepted value, Acceptors piggyback the highest *try* for which they last accepted a value, and the corresponding value. If the Proposer manages to collect Promise messages from a majority, i.e., $f+1$ or more, of Acceptors, then it selects the previously accepted value corresponding to the highest received previously accepted *try* number, chosen from the received set of Promise messages. If all Promise messages indicate that there is no such previously accepted value, then the proposer selects the received command from the upper layer as the value to propose. Let *proposed_value* denote this selected value.

Algorithm 1: Baxos Algorithm

```
// Baxos uses Paxos algorithm as its core and uses random
   exponential backoff for handling contention
Init: promised_try ← −1 // highest promised try
accepted_try, accepted_value ← −1, null // highest accept try and value
Proposer: Prepare (proposal_try)
    broadcast(PREPARE(proposal_try))
    start-timer()
end
Acceptor: onReceive PREPARE (proposal_try) from proposer
    if proposal_try > promised_try then
        promised_try ← proposal_try
        send(PROMISE(promised_try, accepted_try, accepted_value)) to proposer
    end
end
Proposer: onReceive majority of PROMISE (promised_try, accepted_try,
 accepted_value)
    end-timer()
    if any accepted_value ≠ null in responses then
        proposed_value ← accepted_value corresponding to highest try among responses
    end
    if all accepted_values are null in responses then
        proposed_value ← new proposal
    end
    broadcast(PROPOSE(proposal_try, proposed_value))
    start-timer()
end
Acceptor: onReceive PROPOSE (proposal_try, proposed_value) from proposer
    if proposal_try ≥ promised_try then
        accepted_try, accepted_value ← proposal_try, proposed_value
        send(ACCEPT(proposal_try, proposed_value)) to proposer
    end
end
Proposer: onReceive majority of Accept (proposal_try, proposed_value)
    end-timer()
    broadcast(DECIDE(proposed_value))
    update(state-machine)
end
on event Timeout
    Random-Backoff()
    Prepare()
end
```

Propose-Accept Upon successfully collecting Promise messages from a majority of Acceptors, the Proposer broadcasts a Propose message piggybacked with the *proposed_try* and the *proposed_value*. An Acceptor accepts a Propose message, if the *proposed_try* is greater than or equal to the highest *try* number that it promised. Upon accepting a Propose message from the Proposer, Acceptors update their *accepted_try* and *accepted_value* variables with *proposed_try* and the *proposed_value*, respectively. and send an Accept message to the Proposer. The Proposer, upon receiving Accept messages from a majority of Acceptors, decides on that value and informs the upper layer about the decision. Finally, the proposer broadcasts a Learn message to inform Learners about the decision.

The Liveness Challenge

The above two-phase algorithm is the core of Synod Paxos, and it achieves obstruction-free but not lock-free termination: If there are multiple concurrent Proposers, then the above algorithm fails to terminate. An example execution where the termination property is not achieved is depicted in Figure 3.1 (b), where replica 1 and replica 2 concurrently send the Prepare messages, without making any progress. In Synod Paxos, upon learning contention (detected by a timeout event), the Proposer retries the Prepare-Promise phase with a *proposed_try* that is strictly greater than its previous *proposed_try* and *promised_try*. However, immediately retrying phase 1 causes further contention.

Addressing contention is where Baxos differs from Synod Paxos: whereas Synod Paxos does not implement a mechanism to deal with contention, Baxos uses REB to address contention. To avoid contention and achieve lock-free termination, the Proposer in Baxos backs off for a random exponential timeout (indicated as Random-Backoff() in Algorithm 1) before retrying again. Figure 3.1 (c) illustrates how Baxos backs off to handle contention.

REB is appealing as a method of handling contention in Synod Paxos due to three main guarantees of REB: (1) robustness, (2) high throughput and (3) resource utilization efficiency [61] as studied in the networking literature. REB enables appointing nodes in non-conflicting timeouts, so that there is only one node utilizing the shared recourse at a given time interval. In this chapter, we ask the question "can REB bring the same advantages to the domain of consensus protocols?". In the next section we present our random back off scheme and explain why it achieves lock-free termination.

3.3.3 REB in Baxos

We aim to achieve two objectives from our REB scheme: (1) Provide lock-free termination by concluding a single Proposer for a consensus choice with asymptotically logarithmic number of failed proposals (retries) and (2) adapt to changing wide area network conditions such as variable latency.

We propose a REB scheme, called **Baxos REB**, that achieves the two goals above. In the Baxos REB scheme, upon facing l retries, each node first selects a number $k \in (0,1) \subseteq \mathbb{Q}$ uniformly at random. Then each node backs off for $k \times 2^l \times 2 \times RTT$ time period where RTT is the maximum network round-trip time between any pair of replicas (network diameter) (note that Δ is the upper bound of RTT/2). Note that we use $2 \times RTT$ in our backoff time calculation, because there are two network round trips to commit a single request (Prepare-Promise and Propose-Accept) and to allow another proposer to successfully propose a command, other replicas should backoff a minimum of $2 \times RTT$. A proposer, upon successfully proposing a value, l is decreased by one. As shown in Section 3.3.4, Baxos REB ensures that eventually there exists only one Proposer for a sufficiently large time period $4 \times \Delta$, such that a decision is reached.

3.3.4 Consensus Proof

We now provide a proof for single-choice Baxos, which satisfies the four consensus properties [2]: Validity, Termination, Agreement, and Integrity. Validity, agreement, and integrity directly follow from the Synod Paxos proofs [13] because we use the same core as Synod Paxos. Termination is derived using our REB scheme. This section focuses on the termination proof sketch for single-choice Baxos.

Termination of Baxos holds only after the GST is reached, and when there is an upper bound Δ on the message transmission time between any pair of nodes. If there is only a single proposer for a run of Baxos, the protocol trivially terminates, hence we focus on the case with multiple contending proposers.

If there are multiple competing proposals from different proposers, each node backs off over a time period of length $k \times 2^{l+1} \times 2 \times \Delta$, since $RTT = 2 \times \Delta$ after GST is reached, where $k \in (0,1) \subseteq \mathbb{Q}$ and l is the number of retries. For termination to hold, we need to show that with high probability there exists a time interval of length $2 \times 2 \times \Delta$ in which only a single replica stops backing off and makes its proposal. Figure 3.2 illustrates this scenario.

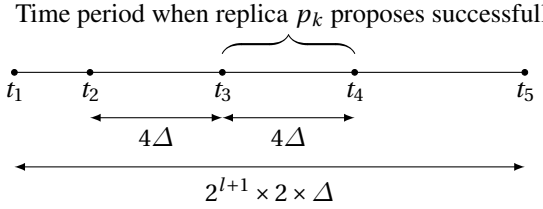


Figure 3.2: Illustration of Baxos termination.

Assume that there are p replicas that compete to propose a value and that each has already done l retries. Assume that all replicas start backing off at time t_1. Let t_5 denote the time at which the last node finishes backing off, then the time interval $(t_1 : t_5)$ has a maximum length of $2^{l+2} \times \Delta$. Depending on k, replica p_k can stop backing off at any time in $(t_1 : t_5)$ and start its proposal phase. Let t_3 denote the time at which proposer p_k stops backing off and starts proposing and

let t_4 denote the time at which p_k successfully decides. The interval $(t_3 : t_4)$ is of length 4Δ which is the duration a proposal requires to complete successfully. Let $(t_2 : t_3)$ denote the interval of length 4Δ before $(t_3 : t_4)$. For p_k to terminate, no other replica should propose in $(t_2 : t_4)$ of length $2 \times 4 \times \Delta$ since any replica which stops its back-off and proposes after t_2 will make its proposal in $(t_3 : t_4)$ putting it into conflict with p_k's proposal. Thus the probability that replica p_k is the only proposer in $(t_2 : t_4)$ is equal to the probability that all other $p-1$ proposers finish their back-offs and proposals in the intervals $(t_1 : t_2)$ and $(t_4 : t_5)$. This probability is given in Equation 3.1.

$$\left(\frac{2^{l+2}\Delta - 8\Delta}{2^{l+2}\Delta}\right)^{p-1} = \left(1 - \frac{1}{2^{l-1}}\right)^{p-1} \tag{3.1}$$

If the value of l is large enough, this probability approaches 1. Hence node p_k eventually succeeds in proposing its value and thus decides.

This proof sketch for termination assumed that each contending proposer starts to backoff at the same time t_1 and that each contending proposer has experienced the same number of retries l. In our experiments, we observed that different proposers start to backoff at different times. For the simplicity of our proof we can let the adversary manipulate the delivery times of messages such that each node starts the backoff timer from the beginning of the synchronized period even if the conflict of replicas is detected at some point $t_1 + t$ where $t < \Delta$ (otherwise a new period starts). Additionally, different replicas have different l, but since the backoffs are exponentially increasing it is obvious that eventually all replicas will reach the same l.

3.3.5 One-Round Trip Optimization

In the absence of leader failures and network partitions, Multi-Paxos consumes a single network round trip time to commit a single client request. This is possible in Multi-Paxos because the leader node runs the Prepare-Promise phase for a sequence of consensus instances, and thereafter, only the leader proposes the commands.

In contrast, Baxos consumes two network round trip delays to commit a single client request, which is a significant drawback. To address this drawback, we apply a classic message piggybacking technique, where the Prepare message for the choice i is piggybacked in the Propose message of choice $i-1$ similar to [70]. Since the Prepare-Promise phase of choice i does not depend on the Propose-Accept phase of choice $i-1$, the Prepare message for choice i can be piggybacked on the choice $i-1$ Propose message. This optimization enables Baxos to commit a request in a single network round trip time, when successive client requests are proposed by the same Proposer. When multiple Proposers propose concurrently, this optimization does not deliver any performance benefit. Given the nature of user interacting web services where a client sends

back to back requests in a partly-open system [71], this design optimization seems a reasonable choice to achieve performance that is comparable to Raft and Multi-Paxos.

3.4 Implementation

We implemented Baxos, Multi-Paxos, and Raft using Golang [72] version 1.15.2. We decided to re-implement Multi-Paxos and Raft in order to have a common framework to compare the performance of these protocols. Had we used the existing implementations of Multi-Paxos [73] and Raft [74], our evaluation would have been influenced by different encoding schemes and different compiler optimizations. We cross-validated our framework implementation by comparing the Multi-Paxos results we obtained against the existing Multi-Paxos implementation[I] by running the experiments using the same setup and workload.

For each consensus algorithm, we used Protobuf encoding [75] and gRPC [76] for message serialization and RPC. We implemented all the attack scenarios we present in this chapter. We did not implement snapshot and replica reconfiguration, which are outside the scope of this chapter.

3.5 Evaluation

The goal of this evaluation is to answer following questions.

- How robust is Baxos against delayed view change attacks in the wide-area networks?
- What is the performance overhead of Baxos during failure-free synchronous periods in the wide-area networks?
- How efficient is Baxos in utilizing bandwidth across replicas in wide-area networks?
- How does Baxos scale with increasing replica count in the wide-area networks?

3.5.1 Experimental Setup

We conducted our experiments using c5d.4xlarge instances (16 virtual CPUs, 32GB memory, and up to 10 Gbps network bandwidth), running Ubuntu Linux 20.04.3 LTS. Each AWS location has a single replica and a single client. Unless mentioned otherwise, we experiment with five consensus replicas and five client replicas ($n = 5$) located in five geographically separated Amazon data centers in N. Virginia, Ireland, N. California, Tokyo, and HongKong.

In Baxos, a client sends requests to the consensus replica in the same data-center as the client; if the server in the same location has failed, then the clients send requests to a randomly chosen

[I] https://github.com/efficient/epaxos/

replica in a different data-center. In Multi-Paxos and Raft, clients send requests to the leader replica. Clients generate requests simultaneously and measure the execution latency for each request. Each experiment was run for 1 minute and was repeated 10 times. We found that longer experiments do not significantly affect the performance results. To amortize the cost of the wide-area network delays, we follow the standard practice of using batching in the replica side with a maximum batch time of 5ms. This resulted in batches of size in the range (5,000, 10,000) requests.

We measure the latency on the client side starting from when a new request is sent by a client until the client receives the response. We set the client request timeout to 8 seconds and requests that took longer than 8s were treated as failed. We measure the throughput on the client side as the ratio of the number of successfully committed requests, excluding failed and timeout requests, and the time duration of the experiment. In the tests where we depict the throughput as a function of time, we aggregate the number of committed requests in one second intervals.

For the delayed view change attack performance results, we changed the transmission delay and packet loss of the replicas using NetEm [77].

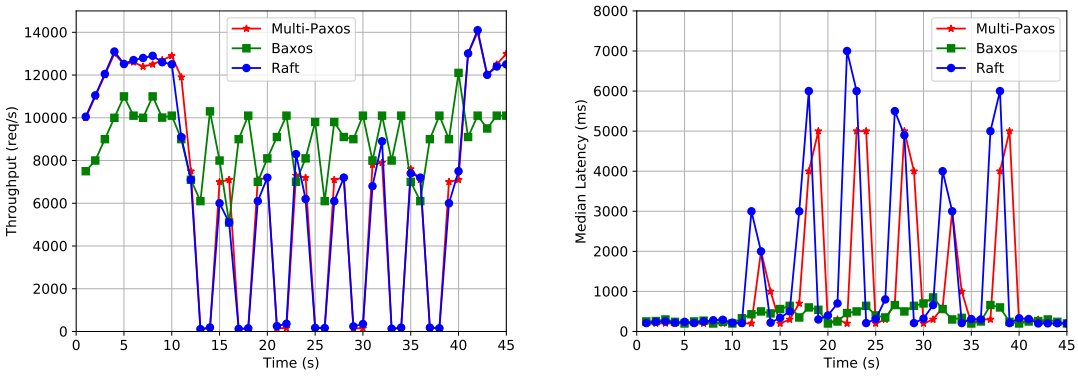
3.5.2 Workload

We use a combination of standard and synthetic micro benchmarks. Our synthetic micro benchmark consists of a configurable service time, configurable request and response sizes. We use the YCSB-A [78] workload with the Redis [79] key value store as the standard benchmark.

Each command in the micro benchmark consists of p bytes of payload and a unique request identifier. All client requests (reads and writes) are totally ordered in Baxos. When a server receives a request, it uses consensus to totally order it, and upon committing and executing, sends a response to client with q bytes with the unique request identifier. After each experiment, we use the replica logs to verify that each replica learns the same sequence of requests. We use open loop model [71] based on the Poisson arrival of client requests for both YCSB-A workload and the synthetic workload.

3.5.3 DDoS Performance

This experiment evaluates the performance of Baxos under adversarial DDoS conditions. The attacker coordinates the attack by adaptively choosing the leader node and attacking it. In Multi-Paxos and Raft, the attacker targets the leader replica and dynamically adjusts the attack by following the current leader upon each view change. In Baxos, there is no designated leader and the attacker chooses an arbitrary replica to attack. We experiment with two types of delayed view change attacks: (1) a *delay attack*, where the adversary increases the transmission delay of a single replica to all destinations and (2) a *packet loss attack*, where the adversary drops a fraction of egress packets of a single replica to all destinations. We used our micro benchmarks for this



(a) Throughput during delay attack (b) Median latency during delay attack

Figure 3.3: Throughput and median latency under delay view change attack

experiment.

We observed the same throughput and median latency variation over time for both delay attack and the packet loss attack. Hence, we only show the delay attack results. Figure 3.3 compares the throughput and median latency of Baxos under delay attack.

We first observe that during the first 10s of the experiment when there is no attack, all three consensus protocols progress at the speed of the network (the best case performance). Second, we observe that the throughput of Multi-Paxos, and Raft fall below 3,500 req/s on average, while Baxos delivers an average throughput of 8,000 requests per second, calculated over the attack time. Fourth, we observe that after 40 seconds (when the attack stops), all consensus algorithms eventually deliver the same performance as before the attack.

We explain the throughput degradation of Multi-Paxos and Raft during the attack period as follows. In the delay attack, the attacker increases the latency of egress packets of the leader in Multi-Paxos and Raft up to 4s. In our experiments, we set the view timeout of Multi-Paxos and Raft to 5s. Since the maximum delay at the leader is less than the view change timeout, each replica receives some messages from the leader before a view change is triggered. To further avoid a view change, the attacker attacks the leader only up to 4s time period in a row, thus giving the leader node the opportunity to perform fast enough without being suspected by the follower nodes as a slow leader. Since the majority of the messages sent by the leader takes 4s on average, this reduces the speed of the entire replica set. This is the reason for observing a low throughput in Multi-Paxos and Raft, during the attack.

Baxos achieves an *average* throughput of 8,000 requests per second even in the presence of attacks. Baxos does not employ a leader replica nor does it depend on the speed of all the nodes; Baxos can make progress at the speed of the majority of replicas. Because the attacker attacks a single random replica at any given time, only the requests which are sent to the replica under attack experience high delays. The impact of the attack is negligible on the other replicas and clients. Hence Baxos delivers a throughput of 8,000 requests per second under attacks, on *average*.

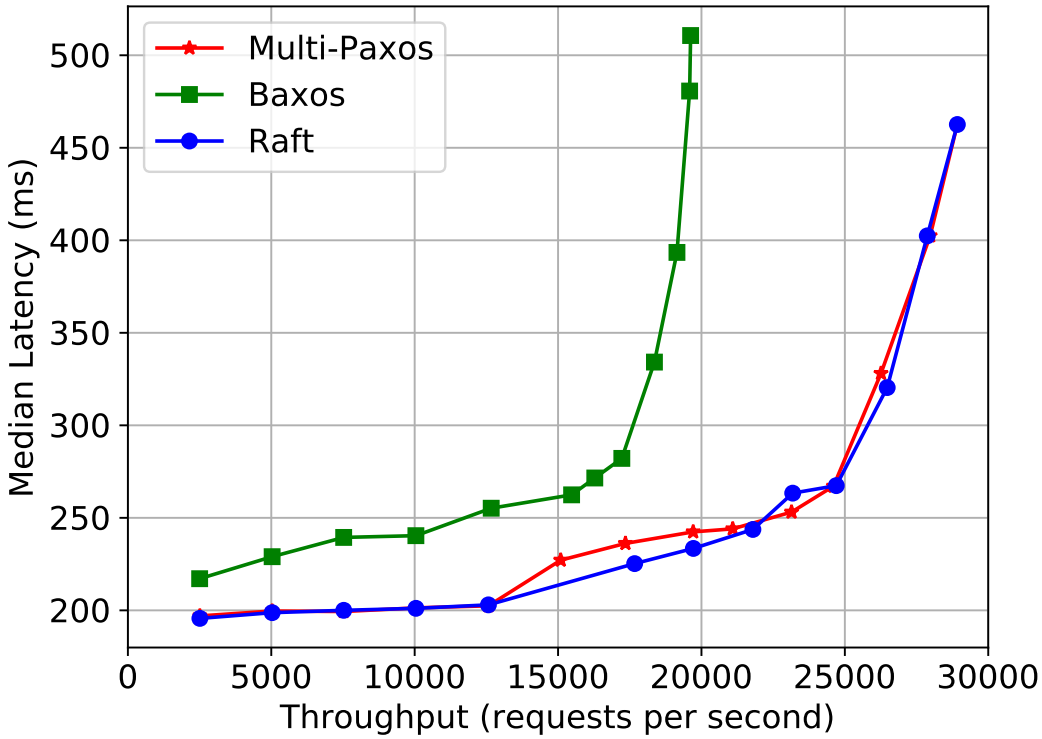


Figure 3.4: Baxos Normal-Case Performance.

Figure 3.3(b) depicts the median latency of each consensus algorithm under study, with respect to time. We first observe that during attack-free executions, all consensus algorithms progress with a median latency less than 200ms. During the attack period (10s-40s), we observe that Multi-Paxos, and Raft deliver a median latency of 1250ms or higher, while Baxos has a median latency of 320ms. The reasoning for this behavior is same as the throughput discussion above: in Baxos the requests that are sent to the attacked replica experiences high delay whereas the requests sent to other replicas do not experience any high delay (thus low overall median latency). In Multi-Paxos and Raft, the latency of each request is affected by the attack given the leader-based message propagation.

We conclude that Baxos is up to 128% more resilient to DDoS attacks in throughput than Multi-Paxos and Raft.

3.5.4 Attack-Free Case Performance

This experiment aims at quantifying the performance overhead of Baxos under faultless and synchronous network conditions. We use five client nodes that simultaneously send traffic to five replicas such that all five replicas propose commands. This experiment measures the worst case performance of Baxos under highest possible contention. Since Baxos must resolve contention

Baxos	Multi-Paxos	Raft
354ms	238ms	235ms

Table 3.1: Tail latency of Baxos

at each choice instead of relying on a stable leader, we expect Baxos to perform worse than leader-based algorithms under stable network conditions, but we wish to measure the performance cost of Baxos's greater robustness. We used our micro benchmark for this experiment.

Figure 3.4 depicts the throughput vs. median latency graph. We observe that for a replica group of size five, Baxos provides a maximum throughput of 17,500 requests per second under 300ms median latency, in contrast to 26,000 requests per second throughput of Multi-Paxos and Raft.

Throughput The saturation throughput of Baxos is 32% less than Multi-Paxos and Raft, because Baxos faces contention: when multiple replicas propose requests simultaneously, their proposals collide, which leads to backing off by replicas and subsequent retries. While Baxos's REB mechanisms enables us to reduce this contention, it cannot completely eliminate its impact. In contrast, Multi-Paxos and Raft do not experience contention because there is a single leader replica which proposes all commands.

Tail latency Table 3.1 illustrates the 99% latency of each algorithm. We observe that the tail latency of Baxos is 48% higher than Multi-Paxos and Raft. The 48% high tail latency of Baxos is caused by the re-transmissions: when Baxos faces contention it re-transmits, whereas in Multi-Paxos and Raft no request is re-transmitted in the best case execution.

These experimental findings show that Baxos provides a low but acceptable performance to Multi-Paxos and Raft in the attack-free and synchronous network settings. We feel that this modest performance cost under high contention is justified in applications such as industrial control systems and remote medical systems, those in which load is sporadic and robustness under all conditions is important. In contrast, if the best case performance is the primary goal, Baxos is appealing as a fallback protocol under DDoS attacks: use Multi-Paxos under default synchronous network settings, and fall back to Baxos if there is a DDoS attack aimed at the leader.

3.5.5 Bandwidth Utilization

Efficiency of resource usage is an important but often overlooked aspect in consensus algorithms [80]. In addition to absolute measures of resources consumed, an efficient consensus algorithm should make each replica spend roughly the same amount of resources [25] resulting in a uniform resource usage. Uniform resource usage is important due to two main reasons: (1) a skew in resource usage results in a higher cost for power in data centers [81] and (2) in resource constrained setups, such as peer to peer systems where each node has the same amount

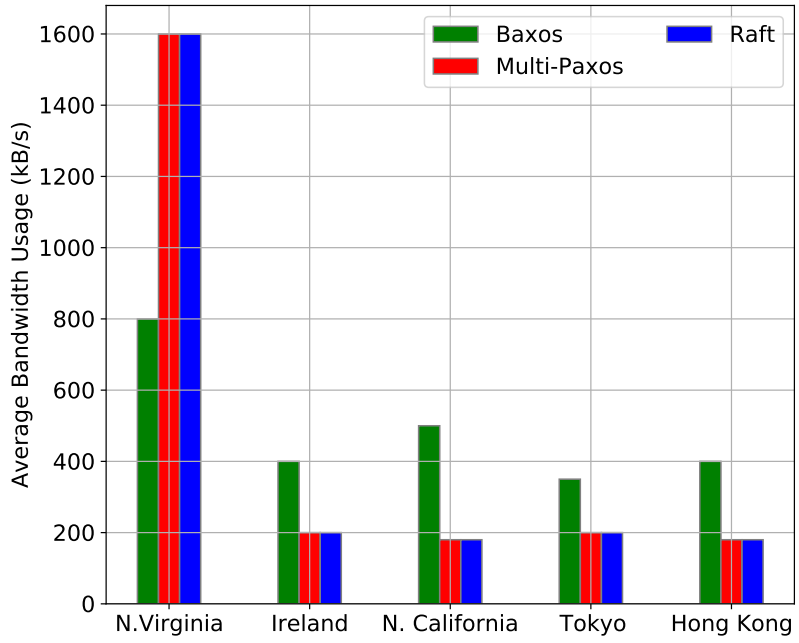


Figure 3.5: Average bandwidth usage of Baxos replicas

of resources, it is prohibitive to have a high resource usage skew. To explore this property, we aim to answer the following question: *What is the variability of resource usage of Baxos replicas running in the wide-area?*

Since we experiment in the wide-area, where the performance is bottlenecked by the speed of the network, we only focus on the network I/O utilization. We use our micro-benchmark for this experiment. To evaluate the variability of the resource utilization by different replicas, we measure the ingress and egress traffic of each replica for a constant arrival rate.

Figure 3.5 depicts the bandwidth utilization of different replicas. For Multi-Paxos and Raft, the leader replica is located in North Virgina. We observe that Multi-Paxos and Raft consume 1,560 kB/s bandwidth on average in the leader replica while consuming less than 200 kB/s in non-leader replicas. In contrast, Baxos consume 220-800 kB/s bandwidth in each replica, thus utilizing the bandwidth more uniformly across replicas. We explain these behaviors as follows.

In Baxos, each replica proposes commands and on average, each replica sends and receives the same amount of messages per second. Hence, in Baxos, each node roughly consumes the same amount of bandwidth. We calculated the standard deviation of the bandwidth utilization of different Baxos replicas to be 152. In contrast, the leader replica in Multi-Paxos and Raft sends and receives more messages than other replicas. This causes Multi-Paxos and Raft to have a bandwidth standard deviation of 560, which is significantly higher than that of Baxos.

While Baxos nodes consume more bandwidth than non-leader nodes in Multi-Paxos and Raft,

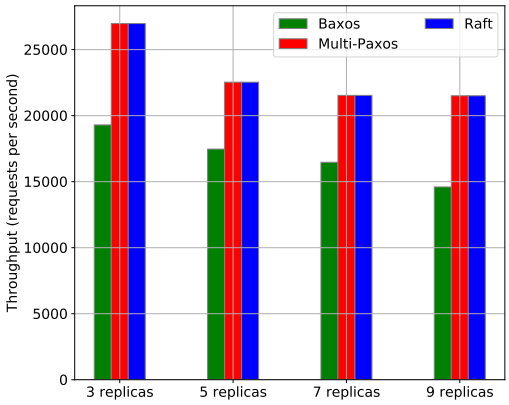


Figure 3.6: Scalability of Baxos with respect to increasing replica count.

its utilization is relatively uniform across nodes and far lower than the leader's bandwidth in leader-based schemes, which makes it a practical choice for data centers and resource constrained deployments such as sensor based internet of things applications.

3.5.6 Scalability in Replica Set Size

This section evaluates the wide-area scalability of Baxos using a standard benchmark. We evaluated the scalability of Baxos, Multi-Paxos and Raft by running them with a replica set size of three, five, seven and nine. Unlike permissioned [18] and permission-less [45] blockchains where consensus algorithms are often designed to scale up to hundreds of nodes, crash fault tolerant protocols are usually designed to scale up to 9 nodes [37][25]. Hence, we evaluate Baxos only up to 9 nodes. We used nine AWS regions located in N. Virginia, Ireland, N. California, Tokyo, HongKong Oregon, Mumbai, Seoul and Cape Town.

We used Redis [79] with YCSB-A [78] workload for this experiment. YCSB-A is a cloud benchmark workload that consists of a mix of 50/50 reads and writes modelling a session store recording recent actions. It assumes 1kB records with 10 fields of 100B each. The key selection is based on the Zipfian distribution. Redis is an in-memory key-value store that supports multiple data structures and operations, such as hash maps, sets and lists. We chose Redis as the backend application due to its wide adoption in the cloud performance analysis literature.

Figure 3.6 depicts the scalability of Baxos with respect to increasing replica counts in the wide-area. We first observe that the throughput values in this experiment are lower than the values presented in Figure 3.4. Second, we observe that when the number of replicas is increased from three to nine, the throughput of each algorithm decreases; the throughput of Baxos decreases from 19,000 to 14,500, whereas for Multi-Paxos and Raft the throughput decreases from 27,000 to 22,500) requests per second.

We observe a reduction of the maximum throughput for all four consensus algorithms compared to the normal case performance experiment above (Figure 3.4) due to the higher network bandwidth usage of this experiment. In the normal case performance experiment, we employed our micro benchmark with a 8B request size whereas in this scalability experiment we employed the YCSB-A workload with a 1kB request size.

The throughput of Baxos, Multi-Paxos, and Raft decrease by 21%, 20%, and 20% respectively, when the replica set size is increased from three to nine, due to two main reasons. First, with increasing replica count, the number of messages sent and received, when proposing a new command by the proposer, increases. Second, with an increasing replica count, the quorum size $(n/2+1)$ increases, thus the proposer has to collect Accept messages from a larger number of replicas. This affects the performance because in the wide-area experiments the proposer has to wait to collect responses from replicas located further away.

Using this empirical study on scalability, we conclude that Baxos scales to a minimum of nine nodes while exhibiting the same percentage throughput loss with respect to the number of nodes as Multi-Paxos and Raft.

3.6 Limitations and Future Work

We now discuss the limitations of Baxos and the future work.

Performance under synchronous executions: As shown in Section 3.5, under synchronous failure free executions, the throughput of Baxos is 32% less than Multi-Paxos and Raft. Moreover, the average and tail latency of Baxos is higher than Multi-Paxos and Raft by 50ms and 116ms, respectively. While Baxos is suitable for applications that require high robustness and moderate performance, Baxos is not suitable for applications that require high performance. Chapter 4 addresses this limitation of Baxos by proposing a novel consensus algorithm that concurrently achieves both (1) high robustness under challenging network conditions and (2) high performance under synchronous network conditions.

Performance under an asynchronous network: Baxos provides robustness against delayed view change attacks, however, it does not provide liveness against an asynchronous network because Baxos assumes partial synchrony for liveness. Hence, under an active network adversary which can arbitrarily order all messages, Baxos loses liveness. This is a main limitation of Baxos, and Chapter 4 solves this limitation using a novel randomized consensus algorithm that provides liveness under an asynchronous network.

Byzantine failures: In this work, we only focus on crash failures. Despite our insights, it might not be straightforward to derive a Byzantine version of Baxos because random backoff is not built on a quorum abstraction. Moreover, malicious parties can lie when they detect the contention and skew their “start backoff time" as they please. In the future, we plan to explore Byzantine Baxos using two approaches: (1) verifiable random functions [82] and (2) trusted hardware base

to enforce random backoff.

Read Optimization: In the current version of Baxos, we do not differentiate between reads and writes, and both reads and writes are totally ordered using the same execution path. We intend to explore read optimizations using *read leases* [24], in the future.

3.7 Related Work

Liveness and performance downgrade attacks. DDoS-resistant protocols based on a "moving target" [21], [83] switch between different approaches depending on the network adversary. When the network is synchronous, these protocols employ single-decree Paxos, which delivers good performance in a synchronous network. When the system is under attack, they employ Ben-Or [26], a randomized asynchronous consensus algorithm. While switching between these protocols provides a good performance when the network is synchronous, it performs poorly (but preserves liveness) when the network is experiencing transient but high delays because of the high message complexity of Ben-Or [26]. Moreover, this approach to DDoS resistance is challenging to implement due to complexities of merging two different consensus protocols. In contrast, Baxos uses the same core consensus algorithm for the attack-free synchronous scenario and the DDoS attack scenario, resulting in fewer lines of code to implement and better performance in the presence of transient high network delays. Spiegelman *et al.* [22] proposed a framework to transform a view based consensus protocol to a randomized consensus protocol to achieve robustness against DDoS attacks. However, their approach has a 100% throughput overhead in the common case (synchronous) execution and as such, it is not suitable for applications requiring a good performance. In contrast, Baxos has only a 32% throughput overhead in the synchronous attack-free execution, compared to Multi-Paxos. Several other works, such as [84], have addressed the robustness of Byzantine consensus protocols under DDoS attacks but assuming a different threat model, where a Byzantine minority of replicas can misbehave. In Baxos, we assumed that replicas are non-Byzantine.

Use of REB and random timeouts in consensus algorithms. Random exponential backoff and random timeouts have been explored in the context of consensus algorithms. IronFleet [85] and PBFT [39] have employed random exponential timeouts to adapt the view change timeout with respect to the network conditions. This allows the replicas to adapt the timeout such that a quorum of Acceptors reply before a view change is triggered. Tendermint [86] employs random timeouts inside a given consensus instance to prevent Tendermint from blocking forever for the liveness condition to be true, and to ensure that processes continuously transition between rounds. Renesse *et al.* [67] use a simmilar approach to increase the time for which a leader waits to collect the responses from Acceptors. Renesse *et al.* [67] employs a TCP-like additive increase, multiplicative decrease approach to select the optimal timeout to wait to collect reposnes from the Acceptors. Raft [14] and Multi-Paxos [13] employs random timeouts to avoid concurrent and contending leader elections. Heterogeneous Paxos [87] employs client side REB to avoid client induced flooding of the system. None of these approaches use REB as the primary method of

contention handling, nor as a mechanism to withstand DDoS attacks. In contrast, Baxos employs REB as the primary method of contention handling to provide resilience against DoOS attacks.

Leaderless consensus algorithms. Mencius [25] achieves consensus without using a leader node by statically partitioning the log space among the set of replicas. This approach has two main drawbacks: (1) the speed of the system is dependent on the slowest replica and (2) an attack on a single replica can negatively affect the overall throughput of the system. In contrast, Baxos makes progress at the speed of the majority of replicas, minimizing the effect of an attack on a single replica on the overall system. Generalized Paxos [56] and EPaxos [24] achieve consensus without a leader by exploiting the request dependencies and using out-of-order commit. These protocols are often more complex than Multi-Paxos, leading to incorrect and complex specifications and implementations [73], [88]. Fast Paxos [89] provides one round trip commits, however, fails to achieve a good performance in the presence of concurrent requests. Both Fast Paxos and Generalized Paxos assume a leader to resolve contention, hence, these protocols do not fully eliminate the leader performance vulnerability. Multi-coordinated Paxos [90] attempts to make the Generalized Paxos leaderless but it fails to deliver a good throughput as compared to Baxos, due to higher message complexity.

3.8 Conclusion

This chapter presented Baxos, the first systematic exploration of the use of random exponential backoff (REB) in place of the usual leader election in Multi-Paxos. Our evaluation shows that Baxos outperforms the commonly used leader-based consensus algorithms such as Multi-Paxos and Raft by 128% in the presence of delayed view change attacks. We also explored the bandwidth efficiency of Baxos and showed that Baxos has a more uniform resource consumption than Raft and Multi-Paxos across replicas. Finally, we showed that Baxos can scale up to nine replicas in the wide-area.

3.8.1 Next Chapter

Baxos has two main limitations: (1) low performance under synchronous executions and (2) liveness loss under asynchronous network conditions, as discussed in Section 3.6. In the next chapter (Chapter 4), we address these limitations by providing a novel consensus protocol that concurrently achieves (1) high performance under synchronous network conditions and (2) liveness under asynchronous network conditions.

4 SADL-RACS: Robust and Scalable Wide-Area Consensus

Most popular consensus protocols deployed in the crash fault tolerant setting, are designed for partially synchronous networks to achieve the lowest latency possible. However, when deployed in the wide-area, they face two key "robustness" challenges. First, they lose liveness when the network is asynchronous. Second, they cannot have a high replication factor because of the high load imposed on the leader-replica making it a bottleneck. This directly restricts the fault tolerance limit "f" based on the desired throughput level. In this chapter, we propose SADL-RACS, a novel modular state machine replication algorithm that addresses these two robustness challenges.

To achieve robustness under asynchronous network conditions, we propose RACS, a novel crash fault-tolerant consensus algorithm. RACS consists of two modes of operations – synchronous and asynchronous – that always ensure liveness. RACS leverages the synchronous network to minimize the communication cost to $O(n)$ and matches the lower bound of $O(n^2)$ at adversarial-case executions. To avoid the leader bottleneck and to allow higher replication factor, without sacrificing the throughput, we then propose SADL, a novel consensus-agnostic asynchronous dissemination layer. SADL separates client command dissemination from the critical path of consensus and distributes the overhead evenly among all the replicas. The combination of RACS and SADL provides a robust and high-performing state machine replication system.

We implement and evaluate SADL-RACS in a wide-area deployment running on Amazon EC2. Our evaluation shows that in the synchronous execution, SADL-RACS delivers up to 500k cmd/sec throughput, in less than 800ms latency, outperforming Multi-Paxos and Rabia by 150% in throughput, at a modest expense of latency. Furthermore, we show that SADL-RACS delivers 196k cmd/sec throughput under adversarial network conditions, whereas Multi-Paxos and Raft lose liveness. Finally, we show that SADL-RACS scales up to 11 replicas with 380k cmd/sec, in contrast to Multi-Paxos's 130k cmd/sec throughput.

The content of this chapter appears in a public preprint available at https://arxiv.org/pdf/2404.04183. This work was done in collaboration with Antoine Desjardins and Eleftherios Kokoris Kogias.

4.1 Introduction

Existing popular consensus protocols face two key robustness challenges in the wide-area network (WAN), because they are designed for partially synchronous networks [13], [14]. First, their liveness is fragile when the network is unreliable, or under a distributed denial-of-service (DDoS) attack [21], [22], [91]. Second, the high load imposed on the leader-replica reduces the number of redundant copies (replication factor) allowed [53], [92], thus limiting the fault tolerance threshold (f).

In WAN, the network conditions can often become adversarial due to transient slowdowns, targeted DDoS attacks [22], or misconfigurations of networks [93]. Under such adversarial conditions, the network delays become unpredictable, hence, the protocols built with partially synchronous network assumptions, fail to provide liveness. The theoretical literature has proposed the use of leaderless randomized protocols [26] that guarantee liveness under adversarial networks, however, the need for $O(n^2)$ message overhead under normal-case synchronous executions is prohibitive for practical deployments. The state-of-the-art randomized protocol Rabia[94] addresses the high latency overhead of Ben-Or[26] and provides high throughput in the low-latency data center context, however, fails to achieve good performance in a wide-area setup, as we show in Section 4.7.1. Finally, the Baxos protocol proposed in Chapter 3 assumes partial synchrony. As a result, Baxos does not guarantee liveness under asynchronous network conditions.

In this chapter, we ask the following question: can we design a protocol that can achieve the best of both worlds: optimal performance under synchronous network conditions and robustness under asynchronous network conditions? Withstanding adversarial network conditions while preserving the synchronous case performance is challenging, because, to preserve the synchronous performance, one has to rely on a leader-based design, whereas, to preserve resiliency against adversarial networks, one has to use randomization. It is non-trivial to merge a leader-based protocol and a randomized protocol, given that these two paradigms assume different network conditions and make different design assumptions.

This chapter proposes **RACS** (**R**esilient **A**synchronous **C**onsensus **S**ystem): a novel randomized consensus protocol that concurrently achieves (1) optimal synchronous case performance and (2) robustness under asynchronous network conditions. RACS enables a leader based one round-trip fast path and provides a randomized fallback path that keeps committing new commands under adversarial network conditions. Unlike Multi-Paxos and Raft that stop committing commands during the view-change phase, RACS's fallback path continues to commit new commands.

Although RACS enables robustness against adversarial networks, RACS alone cannot solve the challenge of increasing the fault tolerant threshold (f), without sacrificing the throughput. Throughput of RACS and existing leader-based protocols such as Multi-Paxos are bottlenecked by the leader replica's available network bandwidth and computational resources. When scaling to more than 3–5 replicas RACS and existing leader-based protocols have to sacrifice the throughput,

as we show in Section 4.7.3, due to increased utilization of resources at the leader-replica. Hence, RACS and existing leader-based protocols face a trade-off between high throughput and high fault tolerance: low replication factor enables high throughput but low fault tolerant threshold, whereas, a high replication factor enables high fault tolerant threshold, but low throughput.

This trade-off is inherent because of the strong coupling between data (commands to be executed) and the consensus messages. In existing leader-based protocols and RACS, consensus messages carry a batch of client commands, such that the size of a consensus message is mostly influenced by the command batch size (sizes in the order of 100KB), whereas the consensus metadata only accounts for a few bytes. Hence, with increasing replication factor, the majority of leader replica's bandwidth is spent for client command dissemination. Hence, the asymptotic linear message complexity of consensus algorithms does not necessarily reflect the experimental performance, when deployed.

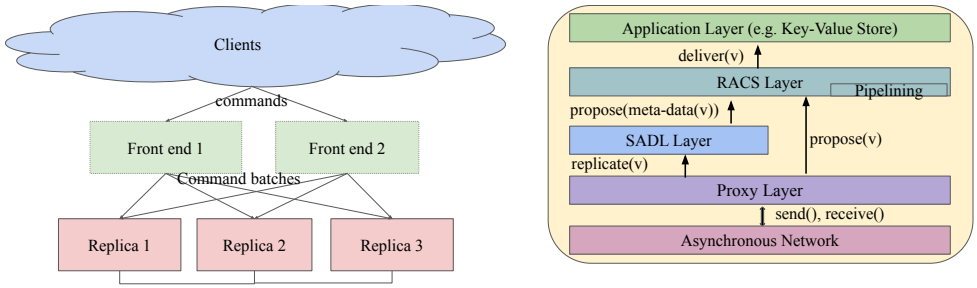
This chapter proposes **S**imple **A**synchronous **D**issemination **L**ayer (**SADL**); a novel asynchronous command dissemination layer. SADL decouples the command dissemination from the critical path of consensus. SADL disseminates client commands, asynchronously, and without employing a designated leader replica, thus distributing the overhead evenly across all the participating replicas. With SADL in place, the leader-replica in RACS only has to send and receive consensus metadata, which are only a few bytes. Hence SADL-RACS can scale to larger number of replicas, without sacrificing throughput.

SADL preserves throughput, but incurs an additional one round trip latency cost, which can be substantial under low replication factors (3 and 5) and under low load, as we empirically show in Section 4.7.1. To accommodate low latency in such cases, we propose a hybrid extension that combines SADL and pipelining, such that applications can dynamically switch between pipelining and SADL depending on the workload and network conditions.

We implemented and evaluated prototypes of RACS and SADL-RACS in Go [72] and compared them against the existing implementations of Multi-Paxos[13], Raft[14], EPaxos [24], and Rabia[94]. We evaluated RACS and SADL-RACS on Amazon EC2 in a multi-region WAN setting. We first show that RACS delivers 200k cmd/sec in throughput under 300ms median latency, comparable to Multi-Paxos's 200k cmd/sec throughput, under synchronous normal case conditions. Second, we show that RACS and SADL-RACS provide 28k cmd/sec and 196k cmd/sec of throughput, respectively, under adversarial network conditions, and outperform Multi-Paxos and Raft which provide 2.8k cmd/sec in the same setting. Finally, we show that SADL-RACS scales up to 11 number of replicas, while delivering a throughput of at least 380k cmd/sec, in contrast, Multi-Paxos sacrifice the throughput with increasing replication factor.

This chapter makes the following key contributions:

- We propose RACS, a novel practical randomized consensus protocol that concurrently provides liveness under adversarial network conditions and high performance under normal case synchronous network conditions.



(a) Architecture (b) Protocol Layering

Figure 4.1: SADL-RACS Architecture and Protocol layering

- We provide formal proofs of RACS.
- We propose SADL, a novel asynchronous command dissemination protocol that enables RACS to support higher fault tolerant threshold (f), without sacrificing the throughput.
- A working prototype and experimental analysis of RACS and SADL-RACS under both normal and adversarial network conditions in AWS.

4.2 Threat Model and Assumptions

We consider a system with n replicas. Up to f (where $n \geq 2f + 1$) number of replicas can crash, but replicas do not equivocate nor commit omission faults [2].

We assume first-in-first-out (FIFO) perfect point-to-point links [2] between each pair of replicas; messages from any correct replica p_i to any correct replica p_j are eventually delivered, in the FIFO manner. We say that a replica broadcasts a message m if it sends m to all n replicas.

We assume a content-oblivious [58] network adversary; the adversary may manipulate network delays, but cannot observe the message content nor the internal replica state. In practice, TLS [95] encrypted channels between each pair of replica satisfy this assumption.

Due to the FLP impossibility result [44], any deterministic algorithm cannot solve consensus under asynchrony even under a single replica failure. In RACS, we circumvent the FLP impossibility result using randomization. In a typical wide-area network, there are periods in which the network behaves synchronously, followed by phases where the network shows asynchronous behavior. RACS makes use of this network behavior and operates in two modes: (1) synchronous mode and (2) asynchronous mode. During the synchronous periods, RACS employs a leader-based design to reach consensus using one round-trip network delay, and during the asynchronous mode, RACS employs randomization. SADL assumes an asynchronous network.

4.3 SADL-RACS Design Overview

4.3.1 SADL-RACS Architecture

Figure 4.1a illustrates SADL-RACS's architecture, containing 2 types of nodes: (1) replicas and (2) front ends. A front end is a node that receives commands from clients scattered elsewhere on the Internet. Front-end node batches commands into client batches and submits them to replicas. Replicas, upon receiving client batches from the front end, form replica batches, and then replicate them in a majority of replicas. Finally, the state machine in each replica executes the totally-ordered commands, and responds to the front end, with the response batches, which are eventually forwarded back to the clients by the front ends. In our discussion, we assume a static set of replicas, however, reconfiguration may be supported via standard practices [14].

4.3.2 Protocol Layering

Figure 4.1b illustrates the protocol layering of SADL-RACS. SADL-RACS consists of 5 layers: the asynchronous network layer, proxy layer, SADL layer, RACS layer, and the application layer. The proxy layer receives client batches from the asynchronous network layer (originating from the front-end nodes). Upon receiving a batch of client batches, the proxy layer forms a replica batch (of size usually more than 100KB) and requests the SADL layer to reliably replicate the replica batch, among at least a majority of the replicas. SADL layer, upon reliably replicating the replica batch, requests the RACS layer to totally order the "meta-data" of the replica batch, which are usually of size several bytes. The RACS layer then total orders replica batch meta-data. Upon reaching consensus, the RACS layer delivers the totally ordered log to the application layer, and the application layer executes the commands in the replica batch. In our implementation, we use Redis[31] and a "map[string]string" key value store as the applications.

As we empirically show in Section 4.7.1, having SADL in the protocol stack reduces the leader bottleneck in the RACS layer, hence, delivers high throughout, and supports high replication factor. Moreover, we show in Section 4.7.2 that, under adversarial network conditions, SADL helps RACS to preserve throughput.

However, under low load, and low replication factor, SADL causes additional latency overhead, hence, may become a burden for the class of applications that require a low replication factor, moderate throughput and low latency. To accommodate such applications, our design allows the proxy layer to directly request the RACS layer (hence bypassing the SADL layer) to totally order the replica batches. When the proxy layer directly invokes RACS, RACS carries the entire replica batch in the critical path of consensus.

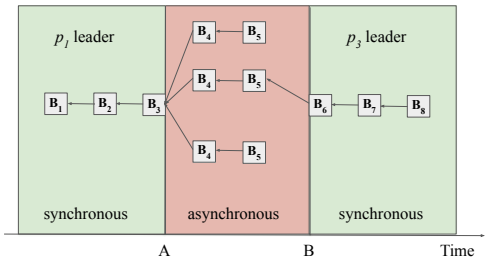


Figure 4.2: An execution of RACS

4.3.3 Robust SMR in the WAN

Given the above operational overview of SADL-RACS, in the following, we summarize how SADL-RACS handles the two robustness challenges we mentioned in Section 4.1.

Challenge 1: Liveness under network asynchrony; Under synchronous network conditions, RACS reaches consensus in one round trip, similar to Raft[14], however, under adversarial network conditions, RACS falls back to a randomized path that keeps committing new requests. Hence RACS provides liveness both under synchronous and adversarial network conditions. SADL does not rely on timeouts for progress, hence provides liveness under adversarial network conditions. Hence, SADL-RACS combination provides resilience to network asynchrony.

Challenge 2: Supporting higher fault tolerant threshold (f) without sacrificing throughput; SADL reliably replicates replica batches (usually of size 100KB) asynchronously, and without relying on a leader replica. With SADL in place, RACS only totally orders meta-data of replica batches, which are usually of size several bytes. Hence, the overhead imposed on the RACS leader is significantly reduced. The low overhead imposed on the RACS leader allows SADL-RACS to scale to a higher number of replicas, without sacrificing the throughput, thus, enabling higher fault tolerant threshold (f).

4.4 RACS

We propose RACS, a novel crash fault-tolerant consensus algorithm that guarantees liveness under adversarial network conditions. RACS employs the chaining approach to SMR similar to Raft [14], in which each new proposal to SMR has a reference to the previous proposal, and each commit operation commits the entire uncommitted history of client commands.

4.4.1 RACS Protocol Overview

RACS has two modes of operation; the synchronous leader-based mode that commits client commands in a single network round trip and a randomized mode that commits client commands under adversarial network conditions. RACS dynamically switches between the synchronous and the asynchronous modes depending on the network condition.

Algorithm 2: RACS Synchronous Path Protocol for replica p_i, $i \in 0..n-1$

Local State:

- $\mathbf{L_v}$ `// designated synchronous leader of view` **v**
- v_{cur}, r_{cur} `// the current view and round numbers`
- $block_{high}$ `// block with the highest rank received`
- $block_{commit}$ `// the last committed block`
- isAsync ← False `// current mode`
- B_{fall}[] `// level 2 asynchronous blocks in the most recent view`
- `argmaxrank(`S`) // returns the block with the highest rank in the messages in` S

Upon *receiving a set S of* $n-f$ *<new-view, v, B> with the same v* ***and*** $v = v_{cur}$ ***and*** *isAsync=False* **do**

- $block_{high} \leftarrow$ `argmaxrank(`S`);`

end

Upon *a new batch of client commands are ready to be proposed* ***and*** *isAsync=False* **do**

- **if** $L_{v_{cur}} == p_i$ **then**
  - cmnds ← getClientCommands();
  - B←(cmnds, v_{cur}, r_{cur}+1, $block_{high}$);
  - v_{cur}, r_{cur} ← B.v, B.r;
  - $block_{high}$ ← B;
  - broadcast <propose, B, $block_{commit}$>;
  - send <vote, v_{cur}, r_{cur}, $block_{high}$> to p_i;
- **end**

end

Upon *receiving <propose, B,* $block_c$*> such that B.rank >* (v_{cur}, r_{cur}) ***and*** *isAsync=False* **do**

- cancel timer();
- v_{cur}, r_{cur} ← B.v, B.r;
- $block_{high}$ ← B;
- $block_{commit}$ ← $block_c$;
- send <vote, v_{cur}, r_{cur}, $block_{high}$> to $L_{v_{cur}}$;
- start timer();

end

Upon *receiving a set S of* $n-f$ *<vote, v, r, B> with the same B* ***and*** *the same (v, r)* ***and*** *rank(B)=(v,r)* ***and*** $v = v_{cur}$ ***and*** *rank(B) > rank(*$block_{commit}$*)* ***and*** *isAsync=False* **do**

- $block_{commit}$ ← B;

end

Upon *local timeout expiration* **do**

- broadcast <timeout, v_{cur}, r_{cur}, $block_{high}$>;

end

Algorithm 3: RACS Fallback Path Protocol for replica p_i, $i \in 0..n-1$

Upon *first receiving a set S of $n-f$ <timeout, v, r, B> messages with the same v such that $v \geq (v_{cur})$* ***and*** *isAsync = False* **do**
- isAsync ← True;
- $block_{high}$ ← `argmaxrank(S)`;
- v_{cur}, r_{cur} ← v, max(r_{cur}, $block_{high}$.r);
- cmnds ← getClientCommands();
- B_{f1} ←(cmnds, v_{cur}, r_{cur}+1, $block_{high}$, 1);
- broadcast <propose-async, B_{f1}, p_i, 1>;

end

Upon *receiving <propose-async, B, p_j, h> from p_j* ***and*** *B.v == v_{cur}* ***and*** *isAsync == True* **do**
- **if** *rank(B) > (v_{cur}, r_{cur})* **then**
  - send <vote-async, B, h> to p_j;
  - **if** *h == 2* **then**
    - $B_{fall}[p_j]$ ← B;
  - **end**
- **end**

end

Upon *first receiving $n-f$ <vote-async, B, h>* ***and*** *isAsync = True* ***and*** *B.v == v_{cur}* **do**
- **if** *h == 1* **then**
  - cmnds ← getClientCommands();
  - B_{f2} ←(cmnds, v_{cur}, B.r+1, B, 2);
  - broadcast <propose-async, B_{f2}, p_i, 2>;
- **end**
- **if** *h == 2* **then**
  - broadcast <asynchronous-complete, B, v_{cur}, p_i>;
- **end**

end

Upon *first receiving a set S of $n-f$ <asynchronous-complete, B, v, p_j>* ***and*** *isAsync = True* ***and*** *v == v_{cur}* **do**
- $L_{elected}$ ← `common-coin-flip(`v_{cur}`)`;
- **if** *level 2 block by $L_{elected}$ exists in S* **then**
  - $block_{high}$, $block_{commit}$ ← level 2 block from $L_{elected}$;
  - v_{cur}, r_{cur} ← rank($block_{high}$);
- **end**
- **else if** *$B_{fall}[L_{elected}]$!= null* **then**
  - $block_{high}$ ← $B_{fall}[L_{elected}]$;
  - v_{cur}, r_{cur} ← rank($block_{high}$);
- **end**
- v_{cur} ← v_{cur}+1;
- isAsync ← False;
- send <new-view, v_{cur}, $block_{high}$> to $L_{v_{cur}}$;
- start timer();

end

Figure 4.2 features an example execution of RACS going from synchronous mode to asynchronous mode, and then to synchronous mode again. After committing 3 blocks, B_1, B_2 and B_3 in the synchronous leader-based mode, with p_1 as the synchronous leader, all replicas fallback to the asynchronous randomized path (due to an adversarial network condition). There, all of the live replicas, namely p_1, p_2 and p_4 propose **exactly** two asynchronous blocks: level 1 block B_4 then a level 2 block B_5. Then, a randomization process called common-coin-flip (see Section 4.4.2) designates p_2 as the elected leader of this asynchronous path. Hence, all replicas commit B_5 proposed by p_2 (and the causal history) and in the following synchronous mode execution commits B_6, B_7 and B_8, with p_3 as the synchronous mode leader.

4.4.2 Preliminaries

RACS progresses as a sequence of views v and rounds r where each view has one or more rounds. A view represents the term of a leader while a round represents the successive log positions in the replicated log. The pair (v, r) is called a **rank**.

There are two kinds of RACS blocks: (1) **synchronous blocks** and (2) **asynchronous blocks**. Both types of blocks consist of five elements: (1) batch of client commands, (2) view number, (3) round number, (4) parent link to a block with a lower rank, and (5) level. The rank of a block is (v, r) and blocks are compared lexicographically by their rank: first by the view number, then by the round number. The blocks are connected in a chain using the parent links. We denote that block A **extends** block B if there exists a set of blocks $b_1, b_2, b_3, ..b_k$ such that there exists a parent link from b_i to $b_{i\text{-}1}$ $\forall$ i in $range(2, k)$ and $b_1 = B$ and $b_k = A$. The level element of the block refers to the asynchronous level (can take either the value 1 or 2). For the synchronous blocks, the level is always 0.

To consistently select a chain in the asynchronous mode of RACS, we use a common-coin-flip primitive. For each view v, common-coin-flip(v) returns a positive integer in the range [0, $n-1$] where n is the total number of replicas. The common-coin-flip(v) satisfies two properties; (1) the invocation of common-coin-flip(v) for a given view v, at each replica should return the same value and (2) output of the invocation of common-coin-flip(i) should be independent of the output of common-coin-flip(j) for $j \neq i$.

Most practical consensus protocols [94], [96] use a common-coin implementation, where the random seed is pre-shared among the replicas. Common-coin assumes a content-oblivious network adversary [58] that cannot observe the message content nor the internal replica state. In this chapter, RACS employs a common-coin where each replica uses a pre-agreed seed to generate the coin values for each view. Upon replica reconfiguration, the old replicas send the coin values to the newly joined replicas. We opted to use a common-coin approach to randomness instead of local-coin based approach, because common-coin based approaches have O(1) expected number of rounds for termination, in contrast to exponential number of rounds in local-coin based approaches [26].

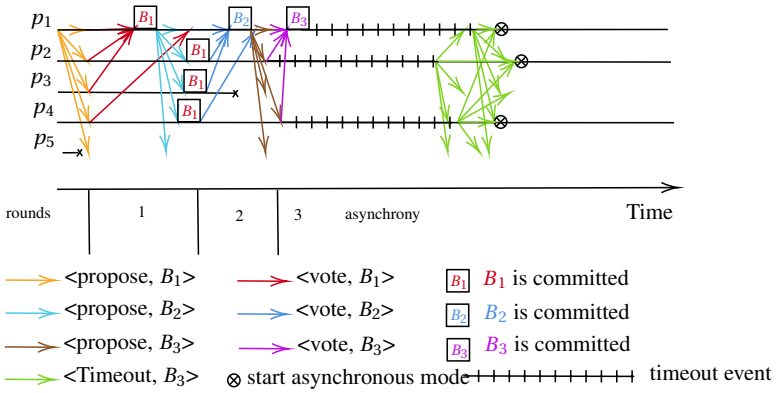


Figure 4.3: Synchronous mode execution of RACS

4.4.3 RACS Algorithm

Algorithm 2 and Algorithm 3 depict the pseudo-code of RACS. In the following discussion, when we say replica p_i delivers a block B from replica p_j, we imply that replica p_i delivers B and the causal history of B.

Synchronous mode: The synchronous mode of RACS is a leader-based consensus algorithm, as depicted in figure 4.3. The synchronous mode leader L_v for each view[I] v is predetermined and known to all replicas on bootstrap.

The synchronous mode begins either at the very start of the SMR or after an asynchronous mode has ended, i.e. upon receiving a majority of <new-view> messages. Upon collecting a sufficient number of client commands, as permitted by the maximum batch size, the leader replica forms a new block B, that extends the $block_{high}$. The leader then broadcasts a <propose> message for the block B containing a rank (v, r) and the reference of the last committed block $block_{commit}$.

Each replica p_i delivers the <propose, B,$block_c$> message, if the rank of B is greater than the rank of p_i and if p_i is in the synchronous mode of operation. If these two conditions are met, then p_i commits $block_c$ ($block_c$ has a lower rank than B) and sends p_i's <vote> for B to the leader replica. Upon receiving $n-f$ <vote> messages for B, the leader replica commits B (and the causal history).

Each replica has a timeout clock which is reset whenever the replica receives a new <propose> message. If the timeout expires, however, they will broadcast a <timeout> message containing the $block_{high}$.

Asynchronous mode: Upon receiving $n-f$ <timeout> messages, RACS enters the asynchronous mode of operation. In the asynchronous mode, all replicas act as leaders, concurrently. Each replica takes the highest $block_{high}$ they are aware of, forms a level 1 asynchronous block B_{f1} with a monotonically increasing rank compared to the highest $block_{high}$ it received and sends a

[I] the synchronous mode leader for view v (L_v) and the elected leader $L_{elected}$ from the common-coin-flip(v) are different.

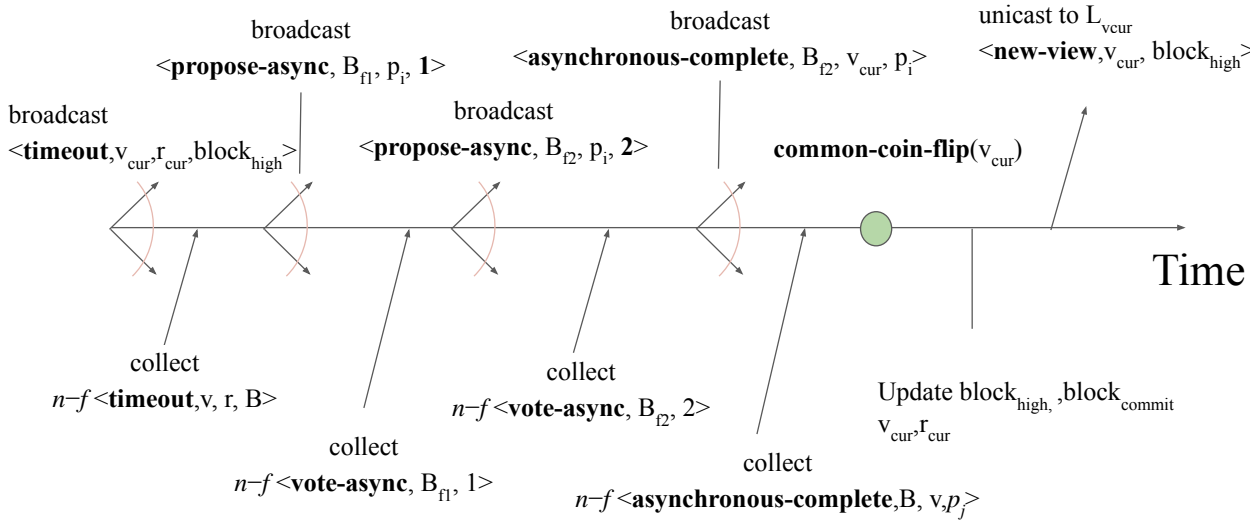


Figure 4.4: An execution of the asynchronous mode of RACS from the perspective of a single process

<propose-async> message.

Upon receiving a <propose-async> message from p_j, each replica p_i sends back a <vote-async> message to p_j **if** the rank of the proposed level 1 block is greater than the highest-ranked block witnessed so far. Upon receiving $n-f$ <vote-async> messages for the level 1 asynchronous block B_{f1}, each replica will send a level 2 asynchronous fallback block B_{f2}. The algorithm allows catching up to a higher ranked block by building upon another replica's level 1 block. This is meant to ensure liveness for replicas that fall behind. All replicas, upon receiving a <propose-async> message for a level 2 asynchronous block from p_j send a <vote-async> message to p_j.

Once $n-f$ <vote-async>s have been gathered for the level 2 asynchronous block B_{f2}, each replica p_i broadcasts an <asynchronous-complete> message. Upon receiving $n-f$ <asynchronous-complete> messages, each replica flips a common-coin to get the elected leader $L_{elected}$. Each replica commits a level 2 asynchronous block B from $L_{elected}$ if B arrived among the first $n-f$ <asynchronous-complete> messages. If a replica observes that the level 2 asynchronous block from $L_{elected}$ does not appear in the first $n-f$ <asynchronous-complete> messages, but appears in $B_{fall}[L_{elected}]$, then, the replica sets $block_{high}$ to $B_{fall}[L_{elected}]$. If a replica observes that the level 2 asynchronous block from $B_{elected}$ is not in the first $n-f$ <asynchronous-complete> messages and $B_{fall}[L_{elected}]$ is null, then the replica doesn't update its $block_{high}$ and $block_{commit}$. After that, all replicas exit the asynchronous path and resume the synchronous path by uni-casting a

<new-view> message to the synchronous leader L_v of the next view with the $block_{high}$.

4.4.4 RACS Formal Proofs

Definition: **elected-asynchronous block**: We refer to an asynchronous block B_f generated in view v with level 2 as an elected-asynchronous block, if the common-coin-flip(v) returns the index of the proposer p_l who generated B_f in the view v and if the <asynchronous-complete> for B_f exists in the first $n-f$ <asynchronous-complete> messages received.

Proof of safety

Theorem 1. *Let B and $\tilde{B}$ be two blocks with rank (v,r). Each of B and $\tilde{B}$ can be of type: (1) synchronous block which collects at least $n-f$ votes or (2) elected-asynchronous block or (3) level 1 asynchronous block which is a parent of an elected-asynchronous block. Then $\tilde{B}$ and B are the same.*

Proof. This holds directly from the block formation – if both B and $\tilde{B}$ has the same rank, then due to quorum intersection, there exists at least one node who voted for both blocks in the same rank, which is a contradiction to our assumption of non malicious nodes. □

Theorem 2. *Let B and $\tilde{B}$ be two adjacent blocks, then $\tilde{B}.r = B.r + 1$ and $\tilde{B}.v \geq B.v$.*

Proof. According to the algorithm, there are three instances where a new block is created.

- Case 1: when $isAsync$ = false and L_v creates a new synchronous block by extending the $block_{high}$ with rank (v,r). In this case, L_v creates a new block with round $r+1$. Hence the adjacent blocks have monotonically increasing round numbers.
- Case 2: when $isAsync$ = true and upon collecting $n-f$ <timeout> messages in view v. In this case, the replica selects the $block_{high}$ with the highest rank (v,r), and extends it by proposing a level 1 asynchronous block with round $r+1$. Hence the adjacent blocks have monotonically increasing round numbers.
- Case 3: when $isAsync$ = true and upon collecting $n-f$ <vote-async> messages for a level 1 asynchronous block. In this case, the replica extends the level 1 block by proposing a level 2 block with round $r+1$. Hence the adjacent blocks have monotonically increasing round numbers.

The view numbers are non decreasing according to the algorithm. Hence Theorem 2 holds. □

Theorem 3. *If a synchronous block B_c with rank (v,r) is committed, then all future blocks in view v will extend B_c.*

Proof. We prove this by contradiction.

Assume there is a committed block B_c with $B_c.r = r_c$ (hence all the blocks in the path from the genesis block to B_c are committed). Let block B_s with $B_s.r = r_s$ be the round r_s block such that B_s conflicts with B_c (B_s does not extend B_c). Without loss of generality, assume that $r_c < r_s$.

Let block B_f with $B_f.r = r_f$ be the first valid block formed in a round r_f such that $r_s \geq r_f > r_c$ and B_f is the first block from the path from genesis block to B_s that conflicts with B_c; for instance B_f could be B_s. L_v forms B_f by extending its $block_{high}$. Due to the minimality of B_f (B_f is the first block that conflicts with B_c), $block_{high}$ contain either B_c or a block that extends B_c. Since $block_{high}$ extends B_c, B_f extends B_c, thus we reach a contradiction. Hence no such B_f exists. Hence all the blocks created after B_c in the view v extend B_c. □

Theorem 4. *If a synchronous block B with rank (v, r) is committed, an elected-asynchronous block $\tilde{B}$ of the same view v will extend that block.*

Proof. We prove this by contradiction. Assume that a synchronous block B is committed in view v and an elected-asynchronous block $\tilde{B}$ does not extend B. Then, the parent level 1 block of $\tilde{B}$, $\tilde{B_p}$, also does not extend B.

To form the level 1 $\tilde{B_p}$, the replica collects $n - f$ <timeout> messages, each of them containing the $block_{high}$. If B is committed, by theorem 3, at least $n - f$ replicas should have set (and possibly sent) B or a block extending B as the $block_{high}$. Hence by intersection of the quorums $\tilde{B_p}$ extends B, thus we reach a contradiction. □

Theorem 5. *At most one level 2 asynchronous block from one proposer can be committed in a given view.*

Proof. Assume by way of contradiction that 2 level 2 asynchronous blocks from two different proposers are committed in the same view. A level 2 asynchronous block B is committed in the asynchronous phase if the common-coin-flip(v) returns the proposer of B as the elected proposer. Since the common-coin-flip(v) outputs the same elected proposer across different replicas, this is a contradiction. Thus all level 2 asynchronous blocks committed during the same view are from the same proposer.

Assume now that the same proposer proposed two different level 2 asynchronous blocks. Since no replica can equivocate, this is absurd. Thus at most one level 2 asynchronous block from one proposer can be committed in a given view change. □

Theorem 6. *Let B be a level 2 elected-asynchronous block that is committed, then all blocks proposed in the subsequent rounds extend B.*

Proof. We prove this by contradiction. Assume that level two elected-asynchronous block B is committed with rank (v, r) and block $\tilde{B}$ with rank $(\tilde{v}, \tilde{r})$ such that $(\tilde{v}, \tilde{r}) > (v, r)$ is the first block

in the chain starting from B that does not extend B. $\tilde{B}$ can be formed in two occurrences: (1) $\tilde{B}$ is a synchronous block in the view $v+1$ or (2) $\tilde{B}$ is a level 1 asynchronous block with a view strictly greater than v. (we do not consider the case where $\tilde{B}$ is a level 2 elected-asynchronous block, because this directly follows from case 2).

If B is committed, then from the algorithm construction it is clear that a majority of the replicas will set B as $block_{high}$. This is because, to send a <asynchronous-complete> message with B, a replica should collect at least $n-f$ <vote-async> messages. Hence, its guaranteed that if $\tilde{B}$ is formed in view v+1 as a synchronous block, then it will observe B as the $block_{high}$, thus we reach a contradiction.

In the second case, if $\tilde{B}$ is formed in a subsequent view, then it is guaranteed that the level 1 block will extend B by gathering from the <timeout> messages B as $block_{high}$ or a block extending B as the $block_{high}$, hence we reach a contradiction. □

Theorem 7. *There exists a single history of committed blocks.*

Proof. Assume by way of contradiction there are two different histories H_1 and H_2 of committed blocks. Then there is at least one block from H_1 that does not extend at least one block from H_2. This is a contradiction with theorems 3, 4 and 6. Hence there exists a single chain of committed blocks. □

Theorem 8. *For each committed replicated log position r, all replicas contain the same block.*

Proof. By theorem 2, the committed chain will have monotonically increasing round numbers. Hence for each round number (log position), there is a single committed entry, and by theorem 1, this entry is unique. This completes the safety proof. □

Proof of liveness

Theorem 9. *If at least $n-f$ replicas enter the asynchronous phase of view v by setting $isAsync$ to true, then eventually they all exit the asynchronous phase and set $isAsync$ to false.*

Proof. If $n-f$ replicas enter the asynchronous path, then eventually all replicas (except for failed replicas) will enter the asynchronous path as there are less than $n-f$ replicas left on the synchronous path due to quorum intersection, so no progress can be made on the synchronous path and all replicas will timeout. As a result, if at least $n-f$ correct replicas broadcast their <timeout> message then all replicas will enter the asynchronous path.

Upon entering the asynchronous path, each replica creates a asynchronous block with level 1 and broadcasts it. Since we use perfect point-to-point links, eventually all the level 1 blocks sent by the $n-f$ correct replicas will be received by each replica in the asynchronous path. At least

$n-f$ correct replicas will send them <vote-async> messages if the rank of the level 1 block is greater than the rank of the replica. To ensure liveness for the replicas that have a lower rank, the algorithm allows catching up, so that nodes will adopt whichever level 1 block which received $n-f$ <vote-async> arrives first. Upon receiving the first level 1 block with $n-f$ <vote-async> messages, each replica will send a level 2 asynchronous block, which will be eventually received by all the replicas in the asynchronous path. Since the level 2 block proposed by any block passes the rank test for receiving a <vote-async>, eventually at least $n-f$ level 2 blocks get $n-f$ <vote-async>. Hence, eventually at least $n-f$ replicas send the <asynchronous-complete> message, and exit the asynchronous path. □

Theorem 10. *With probability $p > \frac{1}{2}$, at least one replica commits an elected-asynchronous block after exiting the asynchronous path.*

Proof. Let leader L be the output of the common-coin-flip(v). A replica commits a block during the asynchronous mode if the <asynchronous-complete> message from L is among the first $n-f$ <asynchronous-complete> messages received during the asynchronous mode, which happens with probability at least greater than $\frac{1}{2}$. Hence with probability no less than $\frac{1}{2}$, each replica commits a chain in a given asynchronous phase. □

Theorem 11. *A majority of replicas keep committing new blocks with high probability.*

Proof. We first prove this theorem for the basic case where all replicas start the protocol with $v = 0$. If at least $n-f$ replicas eventually enter the asynchronous path, by theorem 9, they eventually all exit the asynchronous path, and a new block is committed by at least one replica with probability no less than $\frac{1}{2}$. According to the asynchronous-complete step, all nodes who enter the asynchronous path enter view $v = 1$ after exiting the asynchronous path. If at least $n-f$ replicas never set $isAsync$ to true, this implies that the sequence of blocks produced in view 1 is infinite. By Theorem 2, the blocks have consecutive round numbers, and thus a majority replicas keep committing new blocks.

Now assume the theorem 11 is true for view $v = 0, ..., k-1$. Consider the case where at least $n-f$ replicas enter the view $v = k$. By the same argument for the $v = 0$ base case, $n-f$ replicas either all enter the asynchronous path commits a new block with $\frac{1}{2}$ probability, or keeps committing new blocks in view k. Therefore, by induction, a majority replicas keep committing new blocks. □

Theorem 12. *Each client command is eventually committed.*

Proof. If each replica repeatedly keeps proposing the client commands until they become committed, then eventually each client command gets committed according to theorem 11. □

Complexity The synchronous mode of RACS has a linear message and bit complexity for committing a block. The asynchronous mode of RACS has a complexity of $O(n^2)$.

Having discussed RACS that enables robust consensus over adversarial network conditions, we next present in Section 4.5 the design of SADL, which enables higher fault tolerance, without sacrificing the throughput.

4.5 SADL

SADL is a consensus-agnostic command dissemination layer whose goal is to reliably replicate client command batches. SADL does not solve consensus; rather, it ensures that at least f+1 majority of the replicas are aware of each client command batch that is later proposed by RACS for agreement. The RACS layer does not have to wait for client command dissemination because the SADL is executed ahead of time and concurrently. The RACS layer refers to a chain of SADL command batches using a fixed-sized integer array, making the communication lighter for RACS. Same as RACS, SADL assumes that only up to **f** replicas can crash.

Decoupling command dissemination from consensus has already been explored in the context of blockchain protocols [15] [16]. However, our SADL design significantly differs from existing approaches. Unlike existing decoupling mechanisms where the dissemination layer advances in a lock-step fashion, SADL allows each replica to build its own chain of SADL blocks independently of the pace of other replicas. This key difference enables SADL to achieve higher throughput than existing decoupling approaches. We present this subtle yet non-trivial distinction as a novel systems contribution of this thesis.

4.5.1 SADL Overview

The SADL algorithm runs asynchronously. In SADL, each replica acts as a leader, and disseminates the set of client commands it receives from the front-end servers among a majority of replicas. All replicas run the same algorithmic steps concurrently and do not need to wait for the progress of another replica to move on. This characteristic is what allows the SADL to yield a much higher throughput than RACS.

4.5.2 SADL Algorithm

We introduce **SADL-batch**. A SADL-batch contains four fields: (1) round number, (2) reference to the parent SADL-batch, (3) one or more client commands, and (4) unique identifier.

SADL provides the following generic interface.

- **replicate**(B): replicate a new SADL-batch. We say that the replicate(B) is successful if the replica that replicates B receives at least $n - f$ <SADL-votes> for B.
- **fetch**(B): fetch the SADL-batch corresponding to the identifier of B. A fetch is successful if it returns a SADL-batch B which was successfully replicated previously using replicate(B)

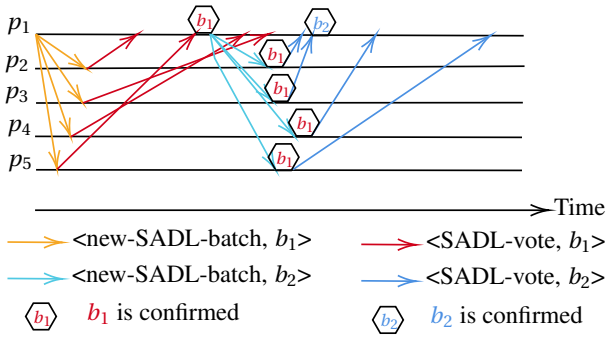


Figure 4.5: An execution of SADL with 5 replicas. All replicas act as leaders, simultaneously. For clarity we only show the execution with only p_1 as the leader

- **fetch_causal(B)**: fetch the set of SADL-batches that causally precede B. fetch_causal(B) succeeds if fetch(B) succeeds and if all the SADL-batches that were replicated before fetch(B) with which B has a causal dependency are in the returned SADL-batch list.

The SADL algorithm has two properties; (1) **Availability**: if fetch(B) is invoked after completing replicate(B), then fetch(B) eventually outputs B and (2) **Causality**: a successful fetch_causal(B) returns all the SADL-batches with which B has a causal dependency.

We explain SADL algorithm using Figure 4.5. In Figure 4.5, we consider a 5 replica setup with p_1 as the sender (in SADL all replicas act as senders, however, for clarity, we only show the execution with only p_1 as the leader). p_1 has received a batch of commands from front-end servers and broadcast them as a <new-SADL-batch> B_1 with round number r_1. All replicas eventually receive B_1 and send back a <SADL-vote> to p_1. Here p_3 and p_4 are slow and send their vote the latest. p_1 receives p_2's <SADL-vote> first, then p_5's. The <SADL-votes> from p_1 (self voting), p_2 and p_5 represent $n - f = 5 - 2 = 3$ votes, so p_1 can consider the round r_1 as completed.

Replica p_1 receives another batch of commands from the front ends and broadcasts a <new-SADL-batch> B_2 to all replicas. B_2 contains the lastCompletedRound[p_1] which indicates that B_1 was completed. Upon receiving B_2, p_2, p_3, p_4 and p_5 learn that B_1 was completed (has received a majority of votes) and update their own $lastCompletedRounds[p_1]$ to B_1. p_2, p_3, p_4 and p_5 send a <SADL-vote> for B_2 to p_1 and the algorithm continues.

4.5.3 Correctness and Complexity

Proof of Availability: A replicate(B) operation succeeds when B is created and sent to all the replicas, and only after receiving at least $n - f$ SADL-votes. Since each replica saves B in the $chains$ array, it is guaranteed that B will persist as long as $n - f$ replicas are alive due to quorum intersection. Hence fetch(B) eventually returns.

Proof of Causality: Causality follows from the fact that each replica extends its chain of SADL-batches, and because each replica creates a batch with round r only after completing the replicate

Algorithm 4: SADL Algorithm for process p_i, $i \in 0..n-1$

Local State:
lastCompletedRounds[] `// last confirmed batch`
chains[][] `//` k^{th} `SADL-batch created by replica` p_j
buffer `// a queue storing incoming client commands`
awaitingAcks ← False
Require : maximum batch time and batch size
Upon *receiving a batch of client commands cl* **do**
  push *cl* to buffer;
end
Upon *(size of incoming buffer reaching batch size* ***or*** *maximum batch time is passed)* ***and*** *awaitingAcks is False* **do**
  B_{parent} ← the SADL-batch corresponding to chains[p_i][lastCompletedRounds[p_i]];
  B ← (lastCompletedRounds[p_i]+1, B_{parent}, buffer.pop());
  awaitingAcks ← True;
  broadcast <new-SADL-Batch, B>;
end
Upon *receiving <new-SADL-batch, B> from* p_j **do**
  chains[p_j][B.round] ← B;
  lastCompletedRounds[p_j] ← B.parent.round;
  send <SADL-vote, B.round> to p_j;
end
Upon *receiving* $n-f$ *<SADL-vote, r> for the same r* ***and*** *r = lastCompletedRounds[*p_i*]+1* ***and*** *awaitingAcks is TRUE* **do**
  awaitingAcks ← FALSE;
  lastCompletedRounds[p_i] += 1;
end
procedure *getClientCommands()***:**
  return lastCompletedRounds;
end

operation of the batch with round $r-1$.

Complexity: The SADL algorithm has a linear complexity: for each batch of client commands, one SADL-batch is broadcast to all replicas and each of these replicas replies to the sender with a <SADL-vote>.

4.5.4 Using SADL with RACS

SADL provides the getClientCommands() interface to the RACS layer. getClientCommands() returns the lastCompletedRounds which contains the last completed SADL-Batch for each replica. Due to the availability and the causality properties of SADL, each SADL batch (and its causal history) indexed by the $lastCompletedRounds[p_i][j]$ is guaranteed to be available in at least a f+1 majority of the replicas. Hence, RACS proposes lastCompletedRounds for agreement among replicas. Since the lastCompletedRounds is a fixed-sized integer array of n elements, the consensus blocks become lightweight. In contrast, in monolithic protocols such as Multi-Paxos, the consensus messages carry the entire batch of client commands, thus sacrificing the throughput.

We evaluated the performance of RACS, under two configurations; (1) RACS with SADL and (2) RACS with pipelining [II]. With SADL enabled, RACS only agrees on the lastCompletedRounds, in contrast, with pipelining enabled, RACS agrees on individual command batches (command batches sent in the critical path of consensus). As shown in Figure 4.6b, we observe that for arrival rates less than 200k cmd/sec, in a 5-replica deployment, the median latency of pipelined-RACS is below 300ms, in contrast, SADL-RACS has 450ms median latency. We also observe that pipelined-RACS cannot sustain arrival rates greater than 250k cmd/sec, in contrast, SADL-RACS delivers stable throughput up to 500k cmd/sec. Hence, we claim that the throughput of SADL-RACS is optimal for higher arrival rates, in contrast, for lower arrival rates, the latency of pipelined-RACS is optimal. Can we achieve the best of both worlds, and have optimal throughput and latency for all arrival rates? Section 4.5.5 addresses this question, by proposing a hybrid pipelined-SADL architecture.

4.5.5 Hybrid SADL-pipelining protocol

The hybrid SADL-pipelining protocol involves two steps: (1) a calibration phase and (2) a deployment phase. In the calibration phase, the system administrator first deploys SADL-RACS and pipelined-RACS protocols, separately, in the given replica and front-end setup, and obtains the throughput versus median latency relationship, which we refer to as the *performance table*.

Then, in the hybrid SADL-pipelining deployment, the RACS layer first starts to replicate command batches using pipelining (without SADL) and the synchronous path leader L_v of RACS monitors the throughput and the median latency. When the median latency reaches

[II] pipelining is a classic technique used in consensus where the leader replica sends block B_i, before getting the $n-f$ votes for the block B_{i-1}

the saturation median latency of RACS according to the *performance table*, the RACS layer automatically proposes a reconfiguration command to enable the SADL instead of pipelining. To consistently enable the reconfiguration across all the participating replicas, we use a similar method to Raft replica set reconfiguration (see [14]: section 6 Cluster membership changes). Once the reconfiguration takes effect, all the replicas switch to SADL. Similarly, if the throughput drops beyond a threshold w.r.t the *performance table*, the RACS leader proposes a reconfiguration change to switch back to pipelining.

With hybrid SADL-pipelining enabled, SADL-RACS delivers the optimal throughput and latency, for all arrival rates.

4.6 Implementation

We implemented RACS and SADL-RACS using Go version 1.18 [72], in 3661 and 4631 lines of codes, respectively, as counted by CLOC [97]. We use the Go network library for TCP [98] and Protobuf encoding [75] for message serialization.

Both RACS and SADL-RACS implement batching in both front ends and replicas as in existing implementations of Rabia [99], Multi-Paxos, Raft[100], and EPaxos [101]. RACS implements pipelining, an optimization available in Multi-Paxos and EPaxos. SADL-RACS does not implement pipelining. Our source code is publicly available[III]

4.7 Experimental evaluation

This evaluation demonstrates the following 4 claims.

- **C1**: Under synchronous network conditions, RACS performs comparably to leader-based algorithms.
- **C2**: RACS offers robustness against adversarial network conditions.
- **C3**: SADL-RACS offers robustness and preserves throughput under adversarial network conditions.
- **C4**: SADL improves the scalability of RACS in two dimensions: (1) **C4.1**: scalability w.r.t increasing replica count and (2) **C4.2**: scalability w.r.t increasing payload size.

Since adversarial networks are much more common in the WAN than in the local-area network (LAN), we focus on the WAN deployments in our evaluation, however, for completeness of experiments, we also compared the performance of RACS in a LAN in Section 4.7.5.

We compare RACS's and SADL-RACS's performance against four state-of-the-art SMR algorithms: Raft [14] (existing implementation [100]), Multi-Paxos [13] (existing implementation [100]), Rabia [94] (existing implementation [99]), and EPaxos [24] (existing implementation

[III] SADL-RACS code: https://github.com/ISTA-SPiDerS/Mandator-Sporades

[101]). Raft is a leader-based chain-replication algorithm based on view-stamped replication [63]. Multi-Paxos is a leader-based protocol that runs the consensus protocol one instance at a time. Rabia is a randomized protocol, that specializes in low latency data center context. EPaxos is a multi-leader protocol that enables parallel commits of non-interfering commands.

Setup

We test both a WAN setup where the replicas and front-ends are distributed globally across AWS regions Sydney, Tokyo, Seoul, Osaka, and Singapore and a LAN setup where all replicas and front-ends are located in North California.

We use Amazon EC2 virtual machines [102] of type t2.xlarge (4 virtual CPUs, 16 GB memory) for replicas and front-ends, for WAN experiments. For LAN experiments, we use instances of type c4.4xlarge (16 virtual CPUs, 30 GB memory) for replicas and front-ends. We use Ubuntu Linux 20.04.5 LTS [103].

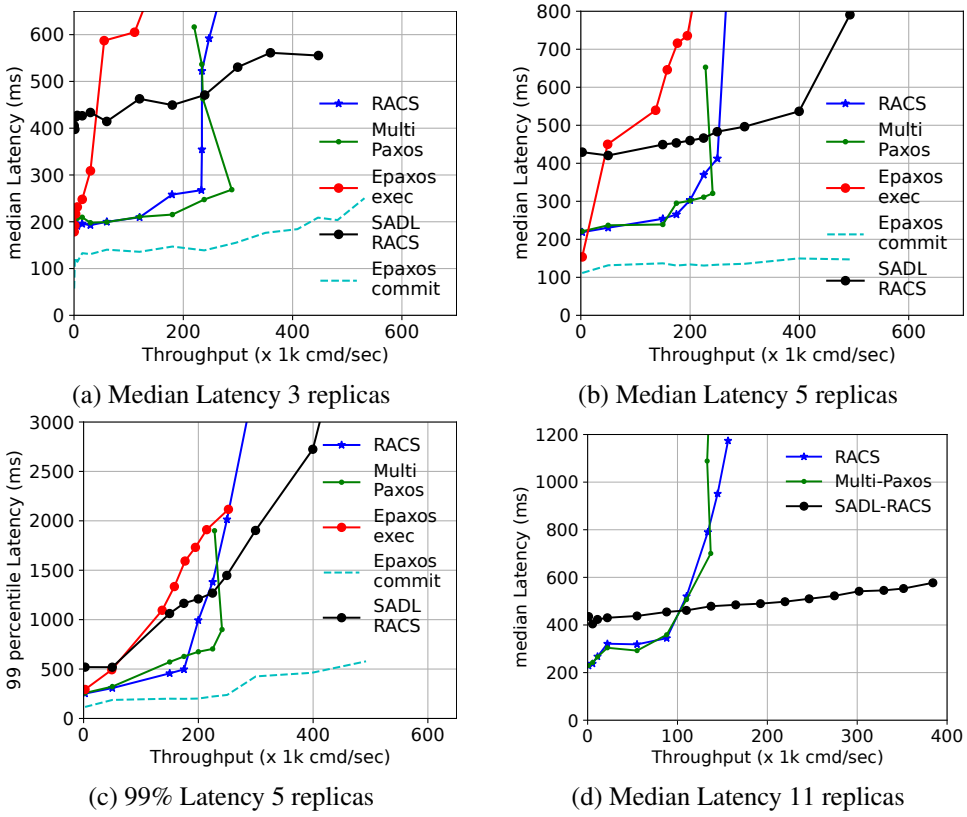
Workload and Benchmarks

Following the existing implementations of Rabia [99], Multi-Paxos, and Raft [100], we use a "map[string]string" key-value store and Redis[31] as backend applications.

In our experiments, we have n replicas and n front ends. Front-ends generate client requests with a Poisson distribution in the open-loop model [71]. All algorithms employ batching in both front-ends and replicas. EPaxos, Multi-Paxos, and RACS support pipelining, while Raft, SADL-RACS, and Rabia implementations do not. A single client request is a 17 bytes string: 1-byte GET/PUT opcode plus 8-byte keys and values, consistent with request sizes used in prior research and production systems [8], [94].

For RACS, SADL-RACS, Multi-Paxos, Raft, and Rabia we measure the front-end observed end-to-end execution latency, which accounts for the latency overhead for total ordering and executing commands. EPaxos provides two modes of operations: (1) partial ordering of commands without execution (denoted “EPaxos-commit” in the graphs) and (2) partial ordering of commands with execution (denoted “EPaxos-exec” in the graphs). Trivially, “EPaxos-commit” outperforms RACS, SADL-RACS, Raft, and Multi-Paxos because EPaxos-commit only provides a partial order of commands, which enables higher parallelism. Hence, “EPaxos-commit” provides an apples-to-oranges comparison, however, we present the results in this evaluation, for completeness. We also found and reported bugs in the existing implementation of EPaxos code that prevent execution under attacks, crashes, and when deployed with more than 5 replicas. Hence, we use EPaxos only under normal-case performance evaluation.

We run each experiment for one minute, repeating it 3 times. We measure throughput in commands per second (cmd/sec), where a command is one 17-byte request. We measure the latency in milliseconds.



(a) Median Latency 3 replicas

(b) Median Latency 5 replicas

(c) 99% Latency 5 replicas

(d) Median Latency 11 replicas

Figure 4.6: Throughput versus latency for WAN normal-case execution, comparing pipelined RACS and SADL-RACS to pipelined Multi-Paxos, and pipelined EPaxos, with 3, 5 and 11 replica ensembles

4.7.1 RACS WAN Normal Case Performance

In this experiment, we evaluate the normal-case synchronous performance of RACS deployed in 5 geographically distant AWS regions. Figure 4.6b and Figure 4.6c depict the experimental results using 5 replicas and 5 front-ends.

RACS vs Multi-Paxos: We observe in Figure 4.6b that RACS delivers a saturation throughput of 200k cmd/sec throughput under 300ms median latency, which is comparable to the performance of Multi-Paxos (200k cmd/sec under 300ms latency). In the synchronous execution both RACS and Multi-Paxos have 1 round trip latency per batch of commands, hence share the same performance characteristics. Hence the experimental claim **C1** holds.

RACS vs Epaxos commit: We observe in Figure 4.6b that EPaxos-commit (without command execution) delivers a throughput of 500k+ cmd/sec under 170ms median latency. The EPaxos-commit experiment employs a conflict rate of 2% [59] hence 98% of the time, commands are committed in one round trip, without serializing through a leader replica. In contrast, RACS builds a total order of commands, serialized using a leader-replica, hence naturally the performance is bottlenecked by the leader replica's capacity.

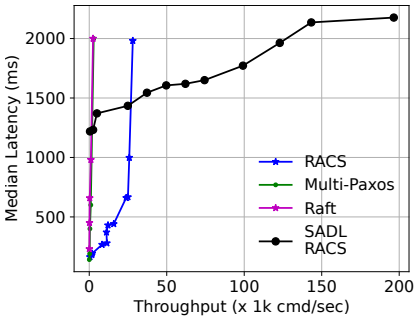


Figure 4.7: Adversarial Performance in the WAN with 5 replicas – note that Multi-Paxos and Raft lines overlap

RACS vs Epaxos exec: As shown in Figure 4.6b, the median latency of EPaxos-exec (with command execution) is 300ms higher on average than RACS in the 50k–200k cmd/sec throughput range. This higher latency stems from EPaxos's dependency management cost [59], [80]. Hence, we conclude that when measured for execution latency, RACS outperforms EPaxos.

RACS vs Rabia: Finally, we observe that Rabia achieves less than 100 cmd/sec throughput under 1000ms median latency (hence not shown in Figure 4.6). Rabia makes a design assumption that network delay is smaller than the interval between two consecutive requests (see section 3.2 of Rabia[94]), a condition that holds only in the LAN deployment. The Rabia paper acknowledges this limitation in their paper and claims performance only in a LAN setting. Section 4.7.5 evaluates Rabia against RACS in a LAN setting.

4.7.2 Asynchronous Performance

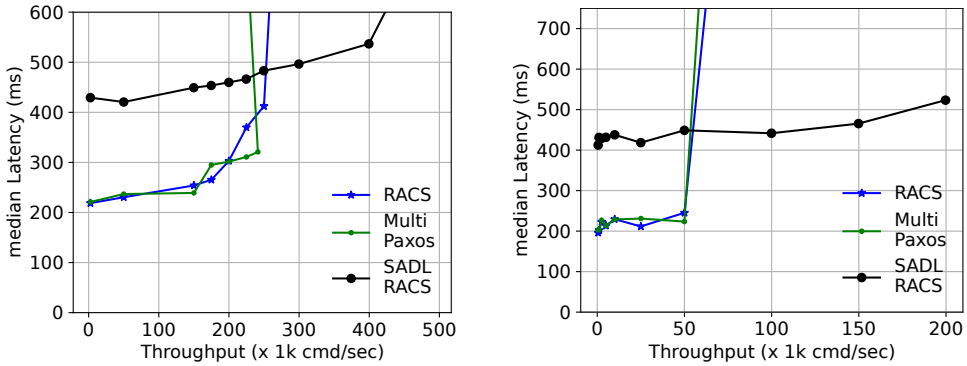
This experiment evaluates RACS and SADL-RACS under simulated network attacks, similar to attacks used in [91], [104]. Our simulated attacker increases the egress packet latency of a minority of replicas chosen at random, by 500ms, dynamically, in time epochs. In each time epoch i, the attacker randomly selects up to f replicas (where $n = 2f + 1$) and launches the attack. This experiment runs in the WAN setting with 5 replicas and 5 front ends. We depict the results in Figure 4.7. [IV] [V]

RACS vs Multi-Paxos:We observe that RACS provides 28k cmd/sec saturation throughput, in contrast, Multi-Paxos and Raft have saturation throughput at 2.8k cmd/sec. Under adversarial network conditions Multi-Paxos and Raft undergo repeated view changes, and fail at successfully committing requests. In contrast, due to asynchronous liveness guarantees, RACS provides liveness under asynchrony. Hence we prove the claim **C2**.

SADL-RACS vs RACS: we observe that SADL-RACS delivers 196k saturation throughput, thus providing 168k cmd/sec more throughput than RACS. SADL disseminates client commands

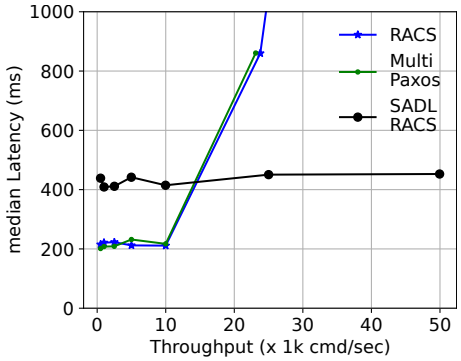
[IV] we do not use Rabia in this experiment, given that Rabia only performs well in the LAN.

[V] we do not employ EPaxos in this experiment because of a dependency management bug we found in the EPaxos code under adversarial network conditions.

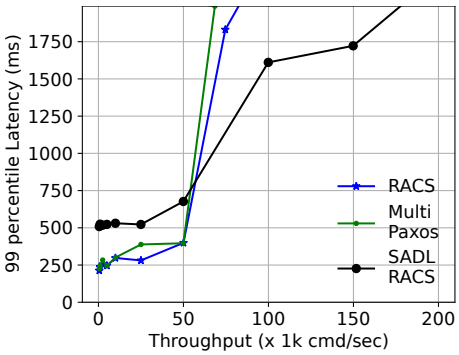


(a) Median Latency (17B)

(b) Median Latency (73B)



(c) Median Latency (265B)



(d) 99% Latency (73B)

Figure 4.8: Throughput versus latency for WAN normal-case execution, comparing SADL-RACS to pipelined Multi-Paxos and pipelined RACS using 17B, 73B and 265B command sizes, using 5 replicas

to a majority of replicas, asynchronously, and in parallel, without relying on a leader. Hence 3 out of 5 replicas in SADL disseminate requests at the speed of the network, in each time epoch, while the other two replicas progress slowly due to the attack, hence SADL-RACS preserves throughput to the maximum level the network conditions allow. This result proves our claim **C3**.

4.7.3 Scalability of SADL

In this experiment, we aim to quantify the scalability of SADL-RACS. We consider two factors of scalability: (1) scalability w.r.t increasing replication factor and (2) scalability w.r.t increasing payload size.

Scalability w.r.t increasing replication factor: In this experiment, we evaluate the scalability of SADL-RACS by running it with an ensemble of three (minimum replication allowed), five (common replication factor) and eleven replicas (improved robustness to concurrent replica

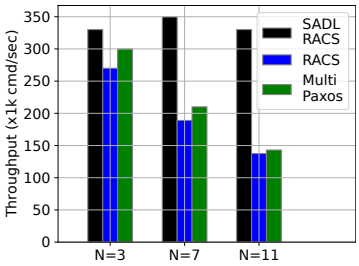


Figure 4.9: WAN scalability with Redis backend

failures), located in geographically separated AWS regions. [VI] [VII] [VIII]. Figure 4.6 compare the scalability of SADL-RACS with pipelined RACS, pipelined Multi-Paxos, and pipelined EPaxos, for different replication factors.

Multi-Paxos and RACS: We observe that the saturation throughput of Multi-Paxos and RACS decreases from 230k to 130k cmd/sec (under 600ms median latency) when the replication factor is increased from 3–11. With increasing replica count, the leader replica in RACS and Multi-Paxos has to send and receive more messages, due to increased quorum sizes, hence the performance is bottlenecked by the leader's bandwidth capacity.

SADL-RACS vs RACS: We observe that SADL-RACS provides a throughput of 380k cmd/sec (under 600 ms median latency), when the replication factor is 11. SADL-RACS outperforms pipelined RACS and pipelined Multi-Paxos by 192% in the 11 replica scenario. This confirms that separating the command dissemination from the critical path of consensus can indeed improve the scalability. Hence we prove the claim **C4.1**.

Scalability w.r.t increasing payload size: In this experiment, we evaluate the impact of payload size for the SADL-RACS performance. We experiment with 3 key sizes: 8B, 64B, and 256B, used in recent SMR work [105]. Combined with 1B opcode and 8B value, these key sizes result in 17B, 73B, and 265B command sizes. We deploy SADL-RACS, pipelined Paxos and pipelined RACS in a WAN setting with 5 replicas and 5 front ends. Figure 4.8 depict the results. [IX]

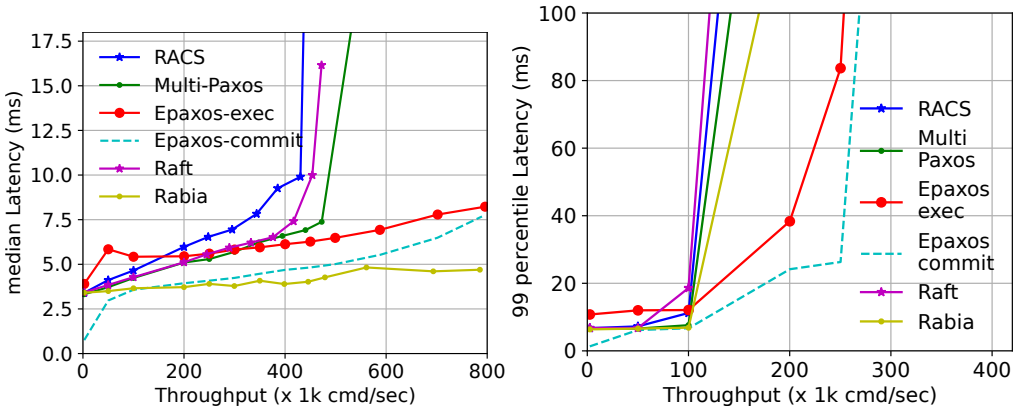
We observe that for each command size, the saturation throughput of SADL-RACS is at least 2 times the throughput of RACS and Multi-Paxos. With increasing command size, the leader replica's bandwidth of RACS and Multi-Paxos becomes the bottleneck. In contrast, thanks to the decoupling of command dissemination from consensus, SADL-RACS evenly distributes the bandwidth overhead among all the replicas, and sustains higher throughput. This proves our final claim **C4.2**.

[VI] Note that, unlike blockchain algorithms where consensus algorithms are measured for up to a hundred nodes[18], [28], crash fault tolerant protocols are designed to scale up to 9–11 nodes in practice [25], [37]

[VII] We did not use Rabia in this experiment, given that Rabia only performs well in the LAN

[VIII] We use EPaxos only in the 3 and 5 replica deployments, due to a dependency-checking bug, we found in the existing code of Epaxos[101] that appears when the number of replicas are greater than 5

[IX] we do not employ EPaxos in this experiment because EPaxos doesn't allow variable sized payloads



(a) Median Latency on LAN

(b) 99% Latency on LAN

Figure 4.10: Throughput versus latency for LAN normal-case execution, comparing RACS to Rabia, Multi-Paxos, EPaxos, and Raft using 5 replicas

4.7.4 Latency overhead of SADL

In Figure 4.6 and Figure 4.8, we observe that SADL-RACS provides better throughput than pipelined RACS, under all replica configurations and under all payload sizes. Moreover, in Section 4.7.2, we observed that SADL-RACS sustains higher throughput than pipelined RACS under adversarial network conditions.

We also observe in Figure 4.6, that for low arrival rates, the median latency of SADL-RACS is higher than the median latency of pipelined-RACS, for instance, in Figure 4.6b for arrival rates less than 200k cmd/sec, the median latency of pipelined-RACS is below 300ms, in contrast, SADL-RACS has 450ms median latency.

To achieve the best of both worlds: (1) high saturation throughput with high robustness against adversarial network conditions and (2) low latency under low arrival rates, we employ the hybrid SADL-pipelined protocol outlined in Section 4.5.5. Our evaluation enables the compilation of the "performance table" of the hybrid SADL-pipelined protocol in Section 4.5.5. We leave the implementation and evaluation of hybrid SADL-pipelining as future work.

4.7.5 RACS LAN Normal Case Performance

We designed RACS and SADL for the WAN, however, for the completeness of the evaluation, we also present the LAN performance. We experiment with 5 replicas and 5 clients, deployed in the same AWS region.

RACS vs Multi-Paxos: We first observe that RACS achieves a saturation throughput of 420k cmd/sec, under a median latency upper bound of 10ms, that is comparable to the saturation throughput of Multi-Paxos (450k) and Raft (440k). Under normal case executions, RACS, Multi-Paxos, and Raft have 1 round-trip latency, serialized through a leader, hence provide comparable

performance.

RACS vs EPaxos: Second, we observe that Epaxos-commit outperforms RACS, both in terms of latency and throughput. We use similar reasoning as Section 4.7.1, to illustrate this behavior.

RACS vs Rabia: Finally, we observe that Rabia delivers 800k cmd/sec throughput under 5ms median latency, outperforming RACS by 380k cmd/sec in throughput, and by 5ms in median latency. Rabia relies on the "natural" ordering of messages inside a data center, where it is guaranteed that a message broadcast at time t will be received by a majority of the replicas within $t + \Delta$. Rabia exploits this data-center-specific "natural" message ordering and uses Ben-Or[26] to check whether the network ordering indeed has achieved consensus (or not). This makes Rabia messages lightweight because the consensus messages no longer have to carry the payload. In contrast, RACS, Multi-Paxos and Raft carry the request payload inside the consensus messages, thus incurring higher latency. Rabia's approach of using consensus to confirm the "natural" ordering inside a datacenter can readily be combined with RACS, Multi-Paxos and Raft, however, falls outside the scope of this work.

4.8 Related Work

Leader based consensus: Multi-Paxos [13], Raft [14], and View Stamp Replication [63] use a single leader node to totally order requests. To reduce the computational and network bottlenecks at the single leader, Baxos [91], Mencius [25], EPaxos [24], Generalized Paxos [56] and Multi-Coordination Paxos [90] use multiple leaders, by partitioning the replicated log using state[24], [56], log position [25] and by using randomized backoff[91]. Leader based and the multi-leader protocols lose liveness under adversarial network conditions, in contrast, RACS maintains liveness under adversarial network conditions. OmniPaxos [106] solves consensus under partial network partitions, a contribution outside the scope of this chapter. OmniPaxos loses liveness under asynchrony. In contrast, RACS maintain liveness under asynchrony.

Asynchronous consensus: Ben-Or[26] is a binary randomized consensus algorithm that provides liveness under asynchrony. Rabia [94] is the first practical multi-valued randomized protocol, that employs Ben-Or as its core. Rabia and RACS achieves orthogonal goals: Rabia provides simplified SMR design for the low latency data center context (hence providing little to no performance in the WAN, and under adversarial network conditions), whereas RACS aims at providing robustness against adversarial networks in both LAN and WAN. Turtle consensus [21] provides robustness to adversarial network conditions by switching between Paxos and Ben-Or, however, provides sub-optimal performance due to static mode switching: turtle consensus lacks the ability to dynamically shift between modes. In contrast, RACS changes mode depending on the network conditions.

Request dissemination Sharding based protocols [54] [33] [38] achieve higher throughput by concurrently committing transactions that touch different shards. Sharding is orthogonal to

SADL-RACS's contributions, and can be readily integrated with SADL-RACS to achieve more concurrency. Overlay based protocols [34], [35], [53], [92], [107]–[109] improve the performance by delegating message propagation to non-leader nodes, however, achieve sub-optimal performance and resiliency against asynchronous networks, compared to SADL. Compartmentalized Paxos [108] reduces the leader bottleneck by using proxy leaders to propagate the payload, however, sacrifices liveness under asynchrony, in contrast, SADL is live under an asynchronous network. Moreover, SADL employs a chaining mechanism to commit a growing sequence of commands in a single consensus instance, thus preserves throughput under asynchrony. In contrast, Compartmentalized Paxos proxy leaders commit one consensus instance at a time, thus achieves lower throughput and lower resilience.

4.9 Conclusion

We presented SADL-RACS, a modular wide-area state-machine replication protocol that achieves robustness under network asynchrony and supports higher replication factors without compromising the throughput. Our evaluation shows that SADL-RACS delivers 500k cmd/sec under 800ms in the wide-area, and out-performs Multi-Paxos by 150%, while remaining live under adversarial network conditions and scaling up to 11 replicas, without sacrificing the throughput.

4.9.1 Next Chapter

In the asynchronous mode of execution, RACS has quadratic message complexity, because of the all-to-all message broadcasting. This overhead is acceptable when facing an active network adversary but becomes unacceptable in most practical adversarial network conditions. In most practical scenarios, the network is not asynchronous, but only transiently slow or has a high loss rate, only for some limited time duration, after which the network becomes synchronous again. Under these practical adversarial scenarios, that last only a few milliseconds/seconds, the quadratic message complexity of RACS is an overkill. In the Chapter 5, we address this limitation by proposing **QuePaxa**, a novel asynchronous consensus protocol, that incurs only linear overhead when the network is only transiently slow. **QuePaxa** still guarantees liveness under an asynchronous network, however, under transient network slowdowns, QuePaxa provides liveness with only linear message overhead.

5 QuePaxa: Escaping the tyranny of timeout in consensus

Most practical consensus algorithms employ a timeout based design to guarantee liveness. We present QuePaxa, a novel protocol offering state-of-the-art normal-case efficiency and robustness to network asynchrony, without depending on timeouts. QuePaxa uses a novel randomized asynchronous consensus core to tolerate adverse conditions, while a one-round-trip fast path preserves the normal-case efficiency of Multi-Paxos or Raft. By allowing simultaneous proposers without destructive interference, and using short hedging delays instead of conservative timeouts to limit redundant effort, QuePaxa permits rapid recovery after leader failure without risking costly view changes due to false timeouts. By treating leader choice and hedging delay as a multi-armed-bandit optimization, QuePaxa achieves responsiveness to prevalent conditions, and can choose the best leader even if the current one has not failed. Experiments with a prototype confirm that QuePaxa achieves normal-case LAN and WAN performance of 584k and 250k cmd/sec in throughput, respectively, comparable to Multi-Paxos. Under conditions such as DoS attacks, misconfigurations, or slow leaders that severely impact existing protocols, we find that QuePaxa remains live with median latency under 380ms in WAN experiments.

The content of this chapter was published in SOSP 2023. This work was done in collaboration with Cristina Basescu, Philipp Jovanovic, Eleftherios Kokoris Kogias, Ewa Syta, Vero Estrada-Galinanes, and Bryan Ford.

5.1 Introduction

Mainstream SMR protocols depend heavily on timeouts in their operation, leading to three related issues we call the *tyranny of timeouts*. First, because most practical consensus algorithms are leader-driven, they rely on partial-synchrony assumptions and timeout-triggered view changes for availability, and may lose liveness under adverse network conditions. Second, because simultaneous leaders interfere destructively and view changes incur high costs, timeouts must be chosen conservatively large to avoid false triggers and maintain performance. Third, timeouts incur the administrative cost of careful manual configuration, and mistakes can lead to poor performance or complete outages. Can practical SMR systems escape the tyranny of timeouts?

We introduce QuePaxa, a novel approach to consensus and SMR that confronts these issues. QuePaxa seeks to ensure robust availability and performance under diverse conditions such as transient slowdowns, targeted denial-of-service attacks, or misconfigurations. In brief, QuePaxa achieves robustness by (1) leveraging randomized asynchronous consensus to guarantee liveness under worst-case conditions; (2) relying on hedging [110], [111] instead of timeouts to gain efficiency comparable to leader-based protocols under normal conditions; and (3) adaptively choosing leaders and hedging schedules to mitigate the costs and risks of manual configuration.

Asynchronous consensus algorithms have long held promise to avoid relying on timeouts for liveness [26], [112], [113], and to tolerate arbitrary network conditions including targeted denial-of-service attacks. Under normal conditions, however, asynchronous algorithms are usually much less efficient than partially-synchronous algorithms. QuePaxa introduces a novel asynchronous crash-stop consensus protocol that randomly prioritizes proposals to circumvent the FLP theorem [44] and guarantees commitment in a few round-trips with high probability. To avoid the traditional efficiency costs of asynchronous consensus, QuePaxa supports a fast path allowing a designated leader to commit in a single round-trip merely by adjusting its priority selection, thereby achieving normal-case efficiency comparable to partially-synchronous protocols.

Timeouts in traditional protocols must be set conservatively large to avoid false triggers, because simultaneous proposers destructively interfere with each other, preventing either from progressing, and unnecessary view changes are costly. Due to QuePaxa's asynchronous core, however, simultaneous proposers not only do not interfere destructively, but can even cooperatively help each other decide faster. In place of timeouts and view changes, proposers in QuePaxa use *hedging* [110], [111] – a delayed-activation schedule – allowing proposers later in the schedule to exercise "enlightened procrastination" to avoid unnecessary effort (computation and bandwidth consumption) redundant with the work of earlier proposers. Hedging enables QuePaxa to achieve the same $O(n)$ complexity as conventional protocols under stable network conditions. Further, hedging delays may be set aggressively small, minimizing recovery time in case of leader failure. False triggers rarely delay QuePaxa consensus, as our experiments confirm, and they never compromise liveness even if hedging delays are badly misconfigured.

Finally, timeouts traditionally incur the administrative cost of manual tuning, and limit the sys-

tem's *responsiveness* [18], [114] or dynamic adaptivity to prevailing conditions. Today's popular consensus protocols such as Raft [14] can remain "stuck" with a leader that is slow but not slow enough to trigger a view change, even when faster leaders are available. QuePaxa incorporates delay monitoring and adaptation techniques, inspired by multi-armed bandit theory [115], to adjust its leader choice and hedging schedule dynamically and ensure responsiveness to time-varying conditions.

The RACS-SADL protocol, as proposed in Chapter 4, addresses the liveness challenges posed by asynchronous networks. However, its performance heavily relies on a timeout mechanism. Specifically, RACS-SADL uses a timeout value to decide when to switch between its two protocol stacks. A poorly configured timeout can result in RACS-SADL operating in the asynchronous mode even when the network remains synchronous. This misconfiguration leads to reduced performance and higher bandwidth consumption due to the quadratic message complexity of the asynchronous path.

In contrast, QuePaxa offers a unified protocol that eliminates the dependency on timeout configurations for performance. By seamlessly handling both synchronous and asynchronous modes within a single protocol framework, QuePaxa avoids the overhead and complexity associated with switching between two distinct protocol stacks. This design ensures consistent performance across network conditions without the drawbacks of timeout-based transitions.

We evaluated a prototype of QuePaxa in Go [72] against Multi-Paxos [13], Raft [14], EPaxos [24], and Rabia [94]. We tested QuePaxa on Amazon EC2 in both local-area (single-region) and wide-area (multi-region) deployments. We first confirm that QuePaxa delivers 584k cmd/sec in throughput under 5.8ms median latency, comparable to state-of-the-art protocols under normal conditions. QuePaxa also remains live, however, under adversarial network attacks that slow or halt existing protocols. QuePaxa maintains full performance even when its hedging delays are only 1/3 the underlying round-trip time, whereas timeouts in Multi-Paxos and Raft must be at least 1.8× the round-trip delay to avoid stalls due to false triggers and unnecessary view changes. Finally, we show that QuePaxa automatically identifies and converges to the fastest leader replica, delivering 1.4ms lower median latency among heterogeneous replicas in a data center setting.

This chapter makes the following main contributions:

- The first protocol that (1) uses hedging instead of timeouts to avoid the high costs of unnecessary view changes, and (2) minimizes recovery time after leader failure.
- Novel adaptation techniques enabling QuePaxa to optimize its leader choice and hedging schedule and maintain responsiveness to prevalent conditions.
- A working prototype and experimental analysis of QuePaxa under both normal and adversarial conditions.
- Correctness proofs of QuePaxa.

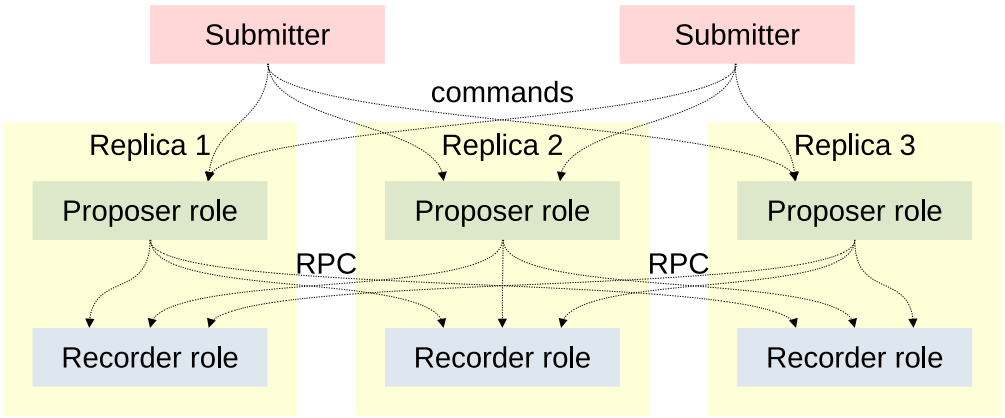


Figure 5.1: Architecture of QuePaxa. A configuration consists of $2f+1$ replicas tolerating f faults. Each replica plays two roles: proposers actively drive consensus, while recorders passively store and update state in response to commands.

5.2 QuePaxa Architecture Overview

This section outlines QuePaxa's architecture at a high level, covering system model, assumptions, workflow, and properties enabling QuePaxa to avoid dependence on timeouts.

5.2.1 QuePaxa system model

Figure 5.1 illustrates QuePaxa's architecture. Any number of *submitters* send *commands* describing transaction requests to a group of *replicas* collectively responsible for storing state. Submitters could be clients that generate commands directly, but in modern tiered deployments, a submitter is more commonly a front-end proxy that receives commands from actual clients elsewhere on the Internet, often gathering commands into batches for submission to the proposers.

As usual in crash-fault consensus, there are $n \geq 2f+1$ replicas, of which at most f may fail, by going silent forever (faults are not Byzantine). We assume the set of replicas is well-known and static, but reconfiguration may be supported via standard practices [13], [14].

Submitters send commands to all replicas, so submitted commands cannot starve indefinitely regardless of which proposers commit transactions. As shown in Figure 5.1, each replica plays two functional roles internally: (1) a replica's *proposer* role receives commands from submitters and actively drives the process of committing these commands, and (2) a replica's *recorder* role passively maintains consensus state while responding to RPC-style requests from proposers. This active/passive division of roles is analogous to that of Disk Paxos [116].

5.2.2 Assumptions and threat model

QuePaxa assumes that while the n replicas are trustworthy, communication paths are not.

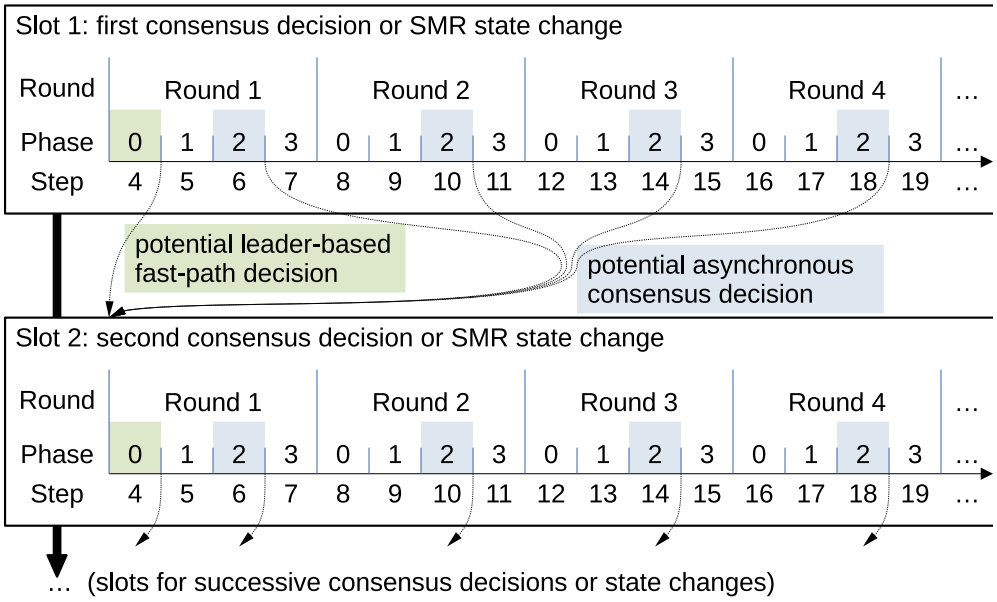


Figure 5.2: QuePaxa workflow overview. *Slots* represent successive state changes decided by consensus. A decision takes one or more *rounds* of four *phases* each. *Steps* combine round and phase numbers into a logical clock: $step = 4 \times round + phase$.

Especially in wide-area networks (WANs), communication may be disrupted in practice by intermittent outages, high delays or jitter, and asymmetric connectivity [93], [117]. Intelligent adversaries in the network may identify consensus protocol flows (e.g. via traffic analysis patterns) and seek to slow or halt progress via targeted denial-of-service (DoS), route hijacking [118], or other attacks.

Formally, QuePaxa assumes that any message sent between correct (non-faulty) nodes is eventually delivered [2], an assumption we fulfill in practice by building atop a reliable transport such as TCP [98]. We consider a *broadcast* to the n replicas to consist of n separate message transmissions in parallel: we do not assume efficient network broadcast.

QuePaxa assumes that the network adversary is content-oblivious [58]. That is, the adversary may manipulate network delays and order packets arbitrarily, but cannot see message content or replica memory. This assumption is realistic in that we can satisfy it in practice simply by encrypting pairwise communication between replicas, e.g. via TLS [95].[I]

5.2.3 QuePaxa workflow overview

Figure 5.2 illustrates QuePaxa's workflow, which employs the standard state machine replication (SMR) paradigm [119]. A series of *slots* represent successive state transitions forming a totally-ordered history. A submitter broadcasts a command (or a batch of commands) to all the proposers, who in turn propose these commands in the next free slot. The proposers then run a consensus protocol instance in each slot to agree on a unique state transition for that slot. Proposers finally report the commitment of commands to the submitter. Proposers may batch concurrent submitters' commands (or batches) together into one slot, or defer them to later slots. Batching is thus possible at both the submitter and proposer stages.

Because asynchronous consensus is deterministically unsolvable in general [44], any single decision attempt may fail and need to be retried. QuePaxa thus decides each slot in a series of attempts or *rounds* numbered from 1. Each round consists of four protocol *phases* 0–3. A *step* number or threshold clock [120] counts phases across rounds, such that $step = 4 \times round + phase$. A step represents progress in completing a threshold of communication, and assumes neither network synchrony nor synchronized clocks. Each step requires one communication round-trip between at least one proposer and a *quorum* or majority of $n - f$ recorders.

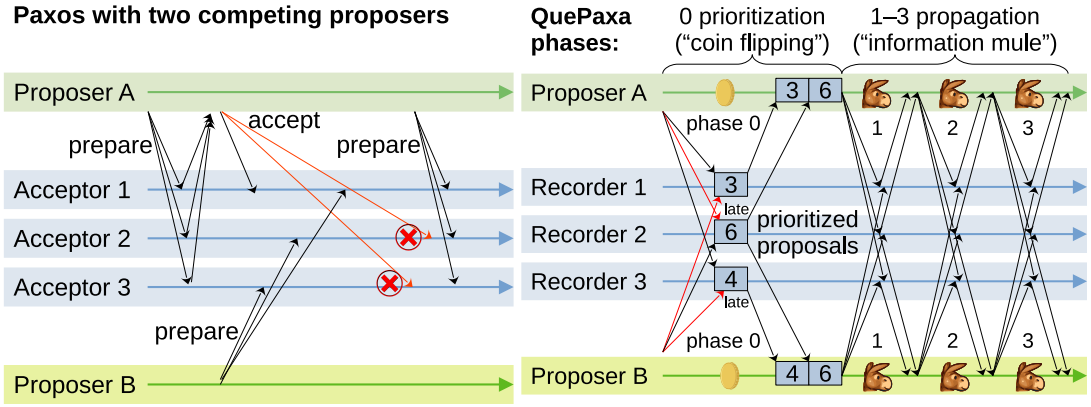
The passive recorder role is simple, merely storing a concise, constant-space summary of recent proposals aggregated via simple arithmetic (e.g. integer maximum). We formulate this recorder functionality in a primitive we call an *interval summary register* or ISR, detailed later in Section 5.3.2.

A proposer may decide a slot in two ways. Under partial synchrony, a unique designated proposer or *leader* may commit in just one round-trip (round 1 phase 0). This fast path is largely equivalent to fast consensus in Multi-Paxos by an already-prepared leader [13]. If the fast path fails for any reason, including leader failure or network asynchrony, then any proposer can potentially decide the slot in phase 2 of any round. Rounds 2 and higher in each slot are leaderless and fully asynchronous, each round guaranteed to succeed independently with probability at least $1/2$. Phase 3 of each round is needed only to prepare for the next round, in case the current round fails to reach consensus.

5.2.4 From competitive claim-staking to cooperation

Like Paxos, QuePaxa is most efficient (and its fast path most likely to succeed) when only one proposer (the leader) proposes at once. In normal-case operation, replicas in both protocols have an expectation of which proposer "should" propose (first) in a slot. The protocols differ fundamentally, however, in the strength of this expectation and in the consequences of its being violated. Figure 5.3 illustrates this difference.

[1] Information leakage via side channels could compromise this content-oblivious adversary assumption. Such risks may be mitigated via implementation best practices such as constant-size messages and constant-time code paths, but side channels are beyond the scope of this chapter.



(a) Destructive interference in Paxos consensus. (b) Cooperative concurrency in QuePaxa consensus.

Figure 5.3: Destructive versus constructive concurrency between multiple proposers in consensus algorithms.

The purpose of Paxos's **prepare** phase is akin to staking a territorial claim on a majority of the proposers, and Paxos's **accept** phase essentially records a successful claim. Two competing proposers interfere destructively, however, in the worst case blocking each other's progress indefinitely. In figure 5.3a, Proposer B's **prepare** phase interrupts Proposer A's attempt to complete (via **accept**) a transaction it had successfully prepared. Proposer A then retries from the **prepare** phase with a higher ballot number, interrupting proposer B – and so on, ad infinitum. This destructive interference motivates traditional view change protocols, in which *only* the leader of a given view can propose – and *must* propose, before the other proposers' timeouts expire – otherwise consensus stalls until further view changes find a live leader. If timeouts are configured too short for the network delay, then exactly this "worst-case" scenario happens and Paxos livelocks forever.

A QuePaxa leader, however, is merely "first among equals" with a special fast-path privilege. Other replicas may also propose, with little risk or penalty apart from redundant effort. Simultaneous proposers do not destructively interfere, and can even help each other complete consensus rounds faster.

As figure 5.3b illustrates, proposers in phase 0 serve a "coin flipping" function, attaching a random priority to the first proposal each recorder sees. In phases 1–3, proposers serve an "information mule" function, propagating information about prioritized proposals among recorders. In both of these functions, it matters little whether just one proposer, or several, perform these steps at once. Consensus rounds complete regardless, each with a constant probability of deciding.

5.2.5 Escaping the tyranny of timeouts

Given the above operational overview, we can now summarize more precisely how QuePaxa escapes from the three "tyranny of timeouts" issues introduced in Section 5.1.

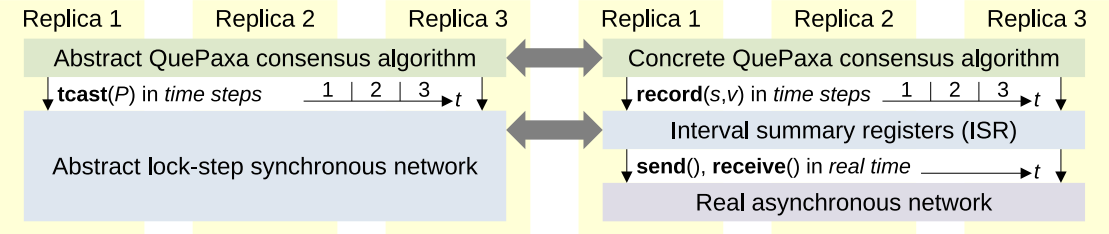


Figure 5.4: Protocol layering. Abstract QuePaxa (Algorithm 5) operates atop lock-step threshold synchronous broadcast (**tcast**), which the concrete protocol (Algorithm 8) simulates using interval summary registers or ISRs (Algorithm 6).

Liveness: QuePaxa eliminates reliance on timeouts for liveness. All phases of all rounds progress asynchronously. The first round in each slot is leader-based, allowing single-round-trip fast-path commit under normal conditions. An intelligent network adversary could potentially exploit knowledge of the leader to make this first round consistently fail to decide, but subsequent rounds are leaderless and fully asynchronous, guaranteeing at least $1/2$ success probability each round.

Hedging: By avoiding destructive interference or view changes, QuePaxa permits all proposers to participate in any round, on a *hedging* schedule [110], [111]. Proposers later in the schedule wait for longer, then propose only if they have not seen earlier proposers make progress. Under normal conditions, only the first scheduled proposer usually proposes in each round, yielding $O(n)$ communication cost as in leader-based protocols. Unlike timeouts, hedging delays may be short for fast recovery after a leader failure. Even if set so small that several proposers start before the first one completes, consensus remains live, the only costs being redundant proposer effort and lower fast-path success probability.

Auto-tuning: Because choice of leader and hedging schedule in QuePaxa are optimization parameters not critical to liveness, these choices are a form of multi-armed bandit (MAB) problem [115]. QuePaxa thus leverages simple explore/exploit processes inspired by MAB theory to explore alternatives and auto-tune consensus to exploit learned knowledge. Unlike existing protocols, QuePaxa can thus find a better leader proactively even if the current leader has not failed. QuePaxa also eliminates both the administrative burden of configuring timeouts and the main risks of misconfiguring them.

Having summarized how QuePaxa operates and avoids dependence on timeouts, we now detail its design.

5.3 QuePaxa Protocol Design

This section details QuePaxa's design, first in terms of a simplified abstract formulation of the core protocol for clarity, followed by a concrete instantiation of that core (see Figure 5.4).

5.3.1 Abstract QuePaxa consensus protocol

For simplicity, we temporarily ignore the challenge of network asynchrony and address *only* the unreliability problem: i.e. that anticipated messages fail to arrive due to replica failures. Figure 5.4 depicts the architectural layers of this abstract protocol, and their correspondence to the concrete protocol we present later in Section 5.3.2. We assume for now that the network provides a *threshold synchronous broadcast* or **tcast** primitive described next. We then define QuePaxa's abstract consensus protocol (Algorithm 5) based on **tcast**. In brief, **tcast** provides each replica with the messages from any majority or quorum of replicas at each time step. Each replica also identifies one message received by all live replicas. Consensus atop **tcast** provides each replica with two proposal sets that bound some set of proposals received by any replica. This cross-node subset relationship is essential for replicas to ensure safety. Each replica attaches a random local priority to its proposal, enabling the protocol to terminate with probability 1 in a small constant expected number of rounds.

Threshold synchronous broadcast (tcast)

We pretend for now that the n replicas run atop an idealized network operating in lock-step synchrony: any message delivery takes exactly one *step*. This idealized network provides a *threshold broadcast* primitive, or **tcast** [113], [120], which we find particularly well-suited to information dissemination in consensus.

At each time step, each live replica i invokes **tcast**(P_i) with some set of proposals P_i that i wishes to disseminate to the other replicas. After one time step, each replica i's **tcast**(P_i) call completes and returns a pair of proposal sets (R_i, B_i). The sets R_i and B_i satisfy two key properties that we define below.

The first set R_i returned by **tcast** is the set of all proposals received by replica i in this broadcast step. This R_i includes the inputs from a majority of replicas. That is, there is some set S of replicas such that $|S| > n/2$, and $\forall\, j \in S, P_j \subseteq R_i$.

The second set B_i returned by **tcast** is some proposal set input (i.e. P_j for some j) that **tcast** has successfully broadcast to *all* non-faulty replicas during this broadcast step. That is, the returned B_i is the proposal set input P_j of some replica j, not necessarily the same as i, such that for all replicas k, $P_j \subseteq R_k$. As a result, for all replicas i and j, $B_i \subseteq R_j$.

In summary, **tcast** ensures two key properties: (1) all live replicas receive a majority of replicas' inputs, and (2) at least one replica's input (returned in B) is seen by *all* live replicas.

Building consensus atop tcast

Algorithm 5 presents QuePaxa's abstract consensus protocol core, built atop **tcast**, for a single SMR slot. Each replica conceptually runs an unlimited series of rounds in the slot, delivering a

Algorithm 5: Abstract QuePaxa consensus algorithm

Input: $v \leftarrow$ value preferred by this replica

```
repeat                                                  // iterate through rounds
    p ← ⟨v, random()⟩                                   // prioritized proposal
    (P, _) ← tcast({p})                                 // propagate our proposal
    (E, P') ← tcast(P)                                  // propagate existent sets
    (C, U) ← tcast(P')                                  // propagate common sets
    v ← best(C).value                                   // next candidate value
    if best(E) = best(U) then                           // detect consensus
        deliver(v)                                      // deliver decision
```

consensus decision with some probability in each round. Different replicas may reach a decision earlier or later, in different rounds.

In each round, each replica i first associates its current preferred value v with a random numeric *priority* to form i's proposal p_i. All replicas choose these priorities independently from the same private random distribution. For simplicity, we will assume for now that priorities never tie within a round.[II]

All n replicas then disseminate their prioritized proposals in three successive **tcast** steps. The first **tcast** gives each replica i a proposal set P_i containing the proposals from any majority of replicas. The second **tcast** takes P_i as input, and gives i a proposal set P'_i that is guaranteed to be included in the *existent* sets E_j returned to all other replicas j. Finally, the third **tcast** uses this set P'_i as input, and gives replica i a *common* proposal set C_i and a *universal* proposal set U_i.

The one important goal that these protocol steps achieve is that $\forall i, j, U_i \subseteq C_j \subseteq E_i$. That is, every replica's universal set U_i is a subset of every other replica's common set C_j, which in turn is a subset of any replica's existent set E_i.

Perhaps more intuitively, a proposal p is *existent* from replica i's perspective (i.e. $p \in E_i$) if i knows that p exists: that is, i knows that some replica proposed p in this round. A proposal p is *common* for i ($p \in C_i$) if i knows that *all* replicas know that p exists. A proposal p is *universal* for i ($p \in U_i$) if i knows that *all* replicas know that p is *common*.

Finally, each replica chooses **best**(C_i), the highest-priority proposal from i's common set C_i, as i's preferred value as input to the next consensus round. Each replica also checks whether its best known existent proposal **best**(E_i) is identical to its best known universal proposal **best**(U_i), and if so delivers this proposal's value as the consensus decision.

[II] We can ensure a negligible chance of a tie for best by choosing priorities with high entropy (e.g. 256 bits) drawn from a strong (e.g. cryptographic) random number generator.

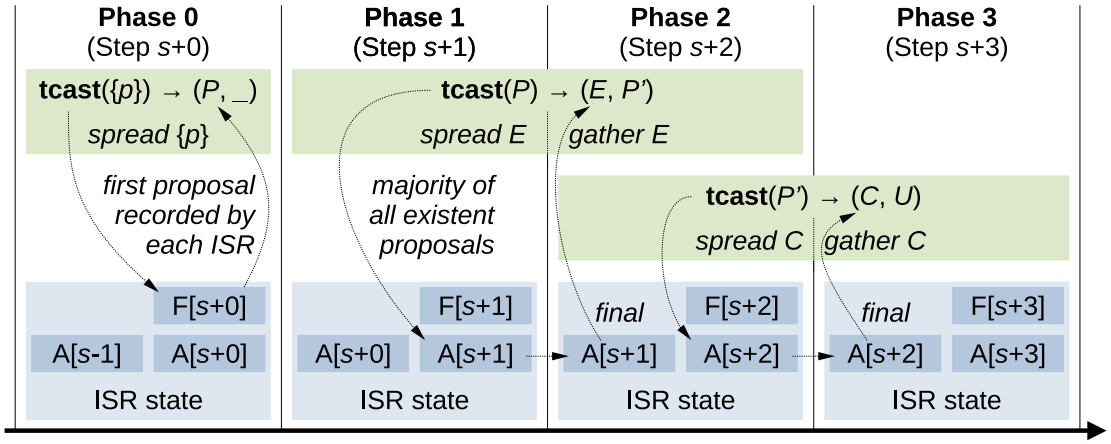


Figure 5.5: Correspondence between the four-phase concrete protocol in Algorithm 8 and the three **tcast** invocations in the abstract QuePaxa in Algorithm 5.

5.3.2 Concrete QuePaxa consensus protocol

The concrete QuePaxa consensus protocol essentially simulates the abstract protocol described above, implementing it more realistically and efficiency in several ways. As outlined earlier in Section 5.2.3, the concrete QuePaxa protocol separates each replica's active and passive roles, handles network asynchrony by threshold logical clocking [120], transmits only constant-space integer summaries rather than proposal sets, and incorporates a Paxos-like fast path to achieve single-round consensus in favorable network conditions with a known leader.

Figure 5.5 shows an overview of how the three **tcast** operations in the abstract protocol above map to the four phases of the concrete protocol detailed below. Concretely implementing the first **tcast** operation in Algorithm 5 requires only one threshold clock time-step (phase 0), because this step requires only that each replica obtain proposals from some majority of replicas. Concretely implementing the second and third **tcast** operations in Algorithm 5 require two threshold clock steps each, using a *spread/gather* sequence detailed below to propagate at least one replica's **tcast** input to *all* live replicas. We can pipeline these latter two **tcast** operations, however, so as to use only three steps total (phases 1–3). The full concrete protocol thus comprises four phases total per round.

Separating active and passive roles

Each replica plays an active *proposer* role, which drives consensus, and a passive *recorder* role, which merely records state. All communication is RPC-style, proposer-to-recorder. Proposers never interact directly with each other, and neither do recorders.

Any proposer can drive consensus, by guiding the recorders through a series of states that simulate

Algorithm 6: Interval summary register (ISR)

```
State : S current logical clock step, initially 0
State : F[s] first value recorded at each step, default nil
State : A[s] aggregate of values in each step, default nil
record (s, v) → (s', f', a'):                          // handle an invocation
    if s > S then                                      // advance to a higher step
        S ← s                                          // update current step number
        F[s] ← v                                       // record first value in this step
    if s = S then                                      // aggregate all values
        A[s] ← aggregate(A[s], v)                      // seen in this step
    return (S, F[S], A[S − 1])                         // return a summary
```

an execution of the abstract QuePaxa protocol (Algorithm 5). As in traditional leader-based consensus protocols, it is sufficient, and most efficient, when only one proposer drives consensus in the common case.

Thus, we expect most replicas' proposer roles (except the leader) to be idle much of the time normally. If multiple proposers *are* active simultaneously, however, they merely work together to drive this simulation faster (i.e. at the speed of the fastest proposer at each step) rather than interfering destructively as in a Paxos-like protocol.

Logical clocks and interval summary registers

As the concrete protocol runs atop an asynchronous network, the recorders use *threshold logical clocks* [120] to simulate the synchronous, lock-step notion of time that Algorithm 5 assumes. Each consensus round consists of four logical time steps. A *step* is a non-negative integer that has no direct correspondence to real time, but advances only when a threshold of communication has been completed in the prior step.

We distill each recorder's state and behavior into a simple abstraction we call an *interval summary register* or ISR, which may be of interest beyond QuePaxa. Intuitively, an ISR accepts a succession of values each associated a logical time step, and in response to each invocation, returns a concise summary of all the values that were presented to the ISR so far in the current and immediately-prior time steps.

Algorithm 6 captures the operation of our ISR in a generic, abstract form. The ISR provides only a single operation, **record**, taking two parameters s, v and returning three results s', f', a'. The value v is associated with logical time-step s.

The **record** operation first uses s to increase the ISR's internal step counter S as needed to the maximum step ever seen so far, and to record the first value v submitted at each step. The ISR then uses some binary combinator **aggregate**, which we will elaborate later in this section, to

Algorithm 7: Specialized constant-space integer ISR

```
State : S current logical clock step, initially 0
State : F_c first value received in current step S, initially 0
State : A_c maximum value seen in this step, initially 0
State : A_p maximum value seen in prior step, initially 0
record (s, v) → (s', f', a'):                          // handle an invocation
    if s = S then                                      // aggregate all values
        A_c ← max(A_c, v)                              // seen in this step
    else if s > S then                                 // advance to a higher step
        if s = S + 1 then                              // exactly one step forward
            A_p ← A_c                                  // current aggregate now prior
        else                                           // skipping one or more step(s)
            A_p ← 0                                    // we saw nothing in s − 1
        S ← s                                          // advance to the new higher step
        F_c ← v                                        // record first proposal this step
        A_c ← v                                        // initial aggregate for this step
    return (S, F_c, A_p)                               // return a summary
```

summarize all values seen during each step. If the step s associated with v is smaller than the ISR's internal step counter S, this means the provided value v is obsolete and the ISR simply discards it. In any case, the ISR returns its internal step counter S, the first value presented in the current step, and the aggregation of *all* values presented in the immediately-prior step.

This ISR formulation assumes that there is a well-defined "base" value we call **nil**, such that **aggregate**$(v,$**nil**$) = v$. Also, for clarity, Algorithm 6 is formulated as if the ISR permanently records values from *all* historical time steps. This is obviously unnecessary, since the ISR only ever returns the first and aggregate values for the current and prior step, respectively. An ISR implementation thus requires only constant space if the values presented to it are constant size.[III]

Specialized ISR for concrete QuePaxa

In the concrete QuePaxa protocol we must instantiate the generic ISR with a suitable value type, **nil** value, and **aggregate** combinator. Since the abstract consensus algorithm (Algorithm 5) uses proposal sets, a naïve ISR for QuePaxa might use a proposal set as its value type, the empty set $\emptyset$ as its **nil** value, and set union $\cup$ as the **aggregate** combinator.

In practice, since we only ever need the **best** or highest-priority proposal in a set, a more optimized implementation of QuePaxa can use simple binary integers as ISR values, zero as **nil**, and integer

[III] A QuePaxa recorder must also store a slot and step number, of course, which might be unbounded in principle. In practice, however, slot numbers may be limited to fixed-size integers by resetting them at reconfiguration events, and forcing a reconfiguration before slot number overflow. Step numbers may be limited to ≈ 10 bits in practice, because the probability of a slot remaining undecided for more than ≈ 256 rounds is cryptographically negligible.

Algorithm 8: Protocol for QuePaxa proposer i

Input: v preferred value of this proposer i

$s \leftarrow 4 \times 1 + 0$ // start at round 1, phase 0
$p \leftarrow \langle H, i, v \rangle$ // initial proposal template
repeat
  $p_j \leftarrow p$ for all recorders j // prepare proposals
  if $s \bmod 4 = 0$ *and* ($s > 4$ *or* i *is not leader)* **then**
    p_j.**priority** $\leftarrow$ **random**$(1..H-1)$ for all j
  Send **record**(s, p_i) in parallel to each recorder j
  Await $R \leftarrow$ quorum of replies (s'_j, f'_j, a'_j)
  if $s'_j = s$ *in all replies received in* R **then**
    if $s \bmod 4 = 0$ **then** // phase 0: propose
      if f'_j.**priority** $= H$ *in all replies* **then**
        return f'_j.**value** *from any reply in* R
      $p \leftarrow$ **best**$_j$ of f'_j from all replies in R
    if $s \bmod 4 = 1$ **then** // phase 1: spread E
      // no action required
    if $s \bmod 4 = 2$ **then** // phase 2: gather E, spread C
      if $p =$ **best**$_j$ *of* a'_j *from all replies in* R **then**
        return p.**value** // report decision
    if $s \bmod 4 = 3$ **then** // phase 3: gather C
      $p \leftarrow$ **best**$_j$ of a'_j from all replies in R
    $s \leftarrow s + 1$ // advance to next step
  else if *any reply in* R *has* $s'_j > s$ **then**
    $s, p \leftarrow s'_j, f'_j$ // catch up to step s'_j

maximum for **aggregate**. A realistic ISR for QuePaxa is thus constant space. For completeness, Algorithm 7 presents pseudocode for the concrete, integer-specialized, constant-space ISR needed by the concrete QuePaxa proposer protocol, which we describe next.

Concrete QuePaxa proposer protocol

Algorithm 8 presents pseudocode for the concrete QuePaxa proposer algorithm. The algorithm uses four logical time-steps per consensus round, starting at step $s = 4$ to represent round 1, phase 0. Figure 5.5 illustrates how these four phases of the concrete protocol correspond to and implement the three **tcast** invocations in Algorithm 5 through interactions with the recorders and their ISR state, as detailed below. Each step incurs one round trip between the proposer and a majority of recorders.

A proposal is logically a $\langle$**priority**, **proposer**, **value**$\rangle$ triple. We assume each component is encoded in a fixed-width binary format, then concatenated, so that the ISR-based recorders

defined above see a proposal simply as a single binary integer.

Because the ISR aggregates values using integer maximum, and **priority** is the first component of the triple, ISR aggregation selects the highest-priority proposal of those submitted during a step, disambiguated by **proposer** in case of a tie.

Proposal randomization: Phase 0 of each round, where $s \bmod 4 = 0$, implements the prioritization of proposals and the first **tcast** in Algorithm 5. Proposer i chooses a random priority on behalf of each recorder, except in leader-based rounds as discussed later in Section 5.3.2. After sending each proposal p_j to recorder j, proposer i awaits replies of the form (s'_j, f'_j, a'_j) from a quorum of recorders. If the step s'_j response from each recorder j in this quorum is equal to the proposer's step number s, the proposer tests for a fast-path decision (Section 5.3.2), then chooses the **best** (highest-priority) proposal from all the f'_j (first-value) responses in the quorum, as i's new proposal p for phase 1 below.

Proposer catch-up: In any phase, if proposer i receives a response (s'_j, f'_j, a'_j) from any recorder j whose $s'_j > s$, this means that proposer i has fallen behind recorder j (and hence behind some other proposer) in logical time. In this case, proposer i simply "catches up" to step s'_j directly, by (a) adopting s'_j as i's new step number s, and (b) taking f'_j as i's proposal template p for this later step.

Spread/gather propagation: Phases 1–3 of Algorithm 8 implement the last two **tcast** invocations in Algorithm 5, in which at least one replica's input is broadcast to *all* live replicas. Algorithm 8 implements these **tcast** operations in two steps each: a *spread* step spreads some proposer's input to a majority of recorders, then a *gather* step gathers knowledge of these spread inputs from a majority of recorders. As Figure 5.5 illustrates, Algorithm 8 pipelines the 2-step spread/gather of existent sets (the second **tcast** in Algorithm 5) together with the 2-step spread/gather of common sets (the third **tcast**), so that these operations take only three steps total in Algorithm 8.

For phase 1 of each consensus round, no phase-specific code is required in Algorithm 8. In this phase, proposer i spreads the "best-of-a-quorum" proposal p resulting from phase 0 (corresponding to the **best** of set P in Algorithm 5) to a quorum of recorders. If proposer i succeeds in this venture, then at the end of phase 1, proposer i knows that the *existence* of its proposal p (or a better one) will become known to *all* proposers by the next phase. This proposal p thus corresponds to the **best** of set P' in Algorithm 5, a proposal set guaranteed to appear in all replicas' existent (E) sets in the abstract algorithm. If proposer i fails to spread its proposal p to a quorum of replicas before some replica advances to the next phase, however, then the generic catch-up logic above leaves i with a (possibly different) proposal that was successfully spread by another (faster) proposer. Either way, i's proposal p at the end of phase 1 is now a *common* proposal.

Phase 2 of each round serves three purposes: to gather knowledge of existent (E) proposals, to spread knowledge of common (C) proposals, and to determine if consensus has been reached. Any proposal that was successfully spread (became common) in phase 1 will have been aggregated

by a majority of recorders' ISRs in phase 1. Any such proposal will thus be accounted for in the prior-step-aggregate a'_j returned by *some* recorder j that i queries in phase 2. The best of a quorum of these aggregates is thus the best of an existent (E) set in Algorithm 5. Further, the working proposal p at the end of phase 2 corresponds to a universal (U) proposal, because the *knowledge that p is common* was spread to a quorum of recorders during this phase. Thus, proposer i can actually perform consensus detection – representing the test if **best**(E) = **best**(U) in Algorithm 5 – at the end of this phase. Proposer i returns a decision immediately if this test succeeds.

Phase 3 is required only in consensus rounds that do not succeed in deciding from proposer i's perspective. In this phase, i gathers knowledge of common (C) proposals, exactly as it gathered knowledge of existent proposals in phase 2. At the end of phase 3, i selects the best of the prior-step-aggregates a'_j of the recorders j in its quorum of responses as its initial proposal p for the next consensus round. This p corresponds to the next candidate **best**(C).**value** computed in Algorithm 5, and defines i's preferred value in the next round.

Fast path: supporting leader-based rounds

The concrete QuePaxa protocol can implement either leaderless asynchronous consensus or efficient leader-based consensus. At the start of each round, all proposers must have already agreed on which proposer, if any, is the leader for this round. This agreement may derive from a prior decision, for example.

In leaderless rounds, all proposers choose the priorities of their proposals as random integers between 1 and $H-1$, where H is the highest possible priority. In this case, no proposer is behaviorally distinguished and QuePaxa acts as an asynchronous consensus protocol in this round.

In leader-based rounds, however, the unique designated leader attaches the highest priority H, reserved for this purpose, to all of its proposals. If the leader's proposal is the first to reach a quorum of recorders in phase 0, then this high-priority proposal naturally dominates the consensus process: *only* the leader's high-priority proposal can ever subsequently be chosen. If the leader obtains such a quorum in phase 0, therefore, the leader can decide at the end of phase 0, after only a single round-trip with the proposers. Under typical network conditions, this fast path enables QuePaxa to commit in a single round-trip, equivalent in efficiency to Multi-Paxos or Raft commitment by an already-prepared leader.

A strong network adversary can always prevent leader-based rounds from succeeding, e.g. by scheduling messages such that the leader's proposal propagates to all proposers' E sets but to none of their U sets. We would thus lose robustness to asynchrony if we always used leader-based rounds. QuePaxa therefore uses a leader only in the *first* round of any slot, then falls back on leaderless rounds if the first round fails to decide. In this way, the leader can generally decide on the fast path in the first round under normal network conditions, while subsequent leaderless

rounds provide a robust asynchronous backup path if the first round fails to decide.[IV]

This leader-based fast path with asynchronous backup addresses our first main "tyranny of timeouts" challenge, liveness loss under network asynchrony. To address the other two challenges, we next focus on how QuePaxa uses the above consensus protocol for state machine replication (SMR).

5.4 High performance SMR with Hedging

This section elaborates on how QuePaxa leverages hedging for efficiency and dynamically optimizes the hedging schedule.

5.4.1 Retroactive versus proactive risk management

Hedging is the practice of launching operations redundantly on different nodes simultaneously, possibly but not necessarily staggered by brief delays, in order to "hedge one's bets" against an instance of the operation taking unexpectedly long [110], [111]. This practice is well-established in large-scale multi-tier query architectures, but to our knowledge QuePaxa is the first work to apply this concept in consensus protocols.

There is a basic difference between a timeout and a hedging delay. A timeout serves to detect a likely failure retroactively, as evidenced by a lack of observed normal-case progress. A timeout typically initiates an abnormal-case recovery process, such as a view change, which interferes with normal-case progress if triggered too early. Hedging initiates *non-interfering* parallel effort, in contrast, proactively limiting risks of long delays. Hedging is safe and often useful even if no failure has occurred. A timeout can never sensibly be configured to zero, as this would leave no time for normal-case progress and would doom the system to an endless failure-recovery loop. A hedging delay of zero not only makes sense but is common, whenever the reduced risk of long delays justifies the costs of simultaneous redundant effort.

5.4.2 Using hedging instead of timeouts in QuePaxa

Leveraging the fact that multiple proposers may be simultaneously active in any protocol step without destructive interference (Section 5.2.4), QuePaxa organizes potential proposers into a *hedging schedule* or delayed-activation sequence. The designated leader, if any, is always the first in the schedule with a delay of zero. All other proposers follow in some known order, sorted in non-decreasing order of associated delays. Each proposer in the schedule waits its associated delay before proposing, and does so *only* if it has not by then seen evidence that some other

[IV] We expect that fast-path optimizations could be pushed further. Using flexible quorums [32], [33], for example, we could reduce the quorum size required in the fast commit path, at the cost of requiring a larger quorum in the following step. We leave such optimizations to future work, however.

proposer (probably earlier in the schedule) has already driven the relevant step to completion.

While the hedging schedule technically needs to include only $f+1$ proposers to ensure liveness against f failures, QuePaxa always includes all proposers in the schedule for simplicity. QuePaxa currently just chooses a single *base delay* parameter δ, then assigns the second proposer (after the leader) a hedging delay of δ, assigns the third proposer a hedging delay of 2δ, etc. Other approaches to scheduling are certainly feasible, such as launching the first two proposers simultaneously, or assigning delays to later proposers based on historically-measured timings of all proposers. We leave such scheduling refinements to future work, however.

During periods of synchrony when the maximum round-trip network delay Δ (unknown to the protocol) is less than the current base delay δ, normally only the first proposer in the schedule will activate, the rest remaining passive upon seeing the leader making progress. Even if δ is less than Δ by some constant factor, however (*i.e.,* $\Delta = O(\delta)$), at most a constant number of proposers will activate in each step, ensuring the same $O(n)$ asymptotic communication cost as in conventional leader-based protocols during synchronous periods. Choosing δ too small can cause too many proposers to activate and revert to the $O(n^2)$ worst-case communication cost that applies under asynchronous conditions. But we will next explore how QuePaxa tunes its choices of leader to prevalent conditions whenever network delays are stable.

5.4.3 Leader tuning in QuePaxa

We usually do not know how each replica will perform as leader until we try it. Even then, observations may be noisy, affected by load and many other factors. Leader selection is thus a form of *multi-armed bandit* problem, a term inspired by a gambling machine ("bandit") whose multiple levers ("arms") each have different, unknown payout chances [115].

QuePaxa adopts a well-known strategy for such problems, first *exploring* or testing alternatives, then *exploiting* or applying learned knowledge. QuePaxa divides SMR slots into fixed-length *epochs* each with a stable leader. In the first $2n+1$ epochs QuePaxa rotates among leaders, round-robin, giving each replica two epochs as leader. After this exploration, QuePaxa exploits these trials by forming and agreeing on a hedging schedule with replicas sorted in descending order of their observed average epoch completion time. QuePaxa then continues monitoring the current leader's performance, recomputing the hedging schedule each epoch, but it no longer proactively explores other leaders unless the current leader's performance falls below that of the next in the schedule.[V]

[V] Refinements inspired by "restless bandits" [121] might periodically re-explore to detect dynamic performance improvements in non-leader replicas.

5.5 QuePaxa Correctness Proofs

5.5.1 Abstract QuePaxa Proofs

In this section, we prove the correctness of abstract QuePaxa described in Algorithm 5.

Definition 5.5.1 (Proposal)**.** We define a *proposal* as a tuple $\langle \pi_i, i, v_i \rangle$, where i denotes the proposer replica's identifier, v_i denotes the proposed value and π_i represents the priority.

This definition adds the replica identifier i to the proposal tuple used in Algorithm 5, to simplify formal reasoning, and in practice for tiebreaking in case there is a non-negligible chance of ties. By this definition, proposals from different replicas are always distinct even if their priorities and values are identical.

In the formal reasoning below, however, we will assume for simplicity that priorities never tie. That is, for any two independently-chosen random priorities π and π', the probability that $\pi = \pi'$ is zero. This is automatically the case if priorities are *real* numbers chosen uniformly at random from the interval between 0 and 1, for example.

Definition 5.5.2 (Time advancement)**.** In the abstract network model that Algorithm 5 builds on, time advances in consecutive integer *steps*. At each time step, each live replica i invokes the *threshold broadcast* primitive, or **tcast**, exactly once. This invocation of **tcast** completes in exactly one time step, unless replica i fails during that time step. Invocation of **tcast** is the *only* way in which time advances in this abstract network model, so all live replicas by definition run **tcast** simultaneously in each time step.

Definition 5.5.3 (Threshold broadcast)**.** At each time step, each live replica i invokes **tcast**(P_i) with some set of proposals P_i that i wishes to disseminate to the other replicas. After one step, if replica i is still live, **tcast** returns to i a pair of proposal sets (R_i, B_i). The **tcast** primitive satisfies two properties, presented informally in Section 5.3.1:

- **Property T1.** R_i includes the inputs from a majority of replicas, *i.e.,* $|R_i| > n/2$, and $\forall P_j \in R_i$, $\exists$ replica j s.t. j proposed P_j in that step.
- **Property T2.** B_i is the proposal set input of some replica (*i.e.,* P_j for some j) that **tcast** has successfully broadcast to all non-faulty replicas during this step. In other words, $\exists$ replica j s.t. $B_i = P_j$, j invoked **tcast**(P_j) in this time step, and $\forall$ live replicas k, $B_i \subseteq R_k$, where (B_k, R_k) is the pair returned from **tcast** on replica k.

Lemma 5.5.1 (Set cardinalities)**.** In every consensus round, the sets P_i, P'_i, E_i, C_i, and U_i that are computed in Algorithm 5 each have cardinality greater than $n/2$.

Proof. By induction over consensus rounds. In the base case, the initial preferred value v_i of each replica i is a well-defined input to the algorithm.

In each round, assuming that v_i is well-defined, the replica's proposal p_i is likewise well-defined, so the input to the first **tcast** invocation is a well-defined proposal set of cardinality one. By **tcast** property T1, the received proposal set P returned from this **tcast** invocation contains the **tcast** inputs to a majority of replicas in this step. Each of these **tcast** inputs has cardinality one and contains a distinct proposal, due both to the inclusion of replica identifier i and to the assumption that priorities never tie, as discussed above. The set P returned from the first **tcast** thus has cardinality greater than $n/2$.

The proposal sets R_i and B_i returned by any **tcast** invocation on replica i are at least as large as the input P_j to **tcast** on *some* replica j in the same time step. As a result, the fact that $\forall$ replicas $j, |P_j| > n/2$ in Algorithm 5 implies that $\forall$ replicas $i, |E_i| > n/2$ and $|P'_i| > n/2$. This in turn implies that $\forall i, |C_i| > n/2$ and $|U_i| > n/2$, As a result, the statement $v \leftarrow \textbf{best}(C_i).\textbf{value}$ chooses a next candidate value from a non-empty common set C_i, thus ensuring that v is well-defined in the next consensus round as well. □

Lemma 5.5.2 (Set relationship)**.** The QuePaxa protocol achieves the property $\forall$ live replicas $i, j, U_i \subseteq C_j \subseteq E_i$ (Section 5.3.1).

Proof. The QuePaxa protocol executes three **tcast** steps. The first **tcast** gives replica i a proposal set P_i containing the proposals of any majority of replicas. The second **tcast** takes P_i as input, and gives replica i a proposal set P'_i that, by **tcast** property T2, is guaranteed to appear in the *existent* sets E_j returned to any replica j. Finally, the third **tcast** uses P'_i as input, and returns to i a *common* proposal set C_i and an *universal* proposal set U_i. By **tcast** property T2, U_i is guaranteed to appear in the sets C_j returned to all replicas j (hence $U_i \subseteq C_j$, $\forall$ replicas i, j). By **tcast** property T1, $\forall P''_j \subseteq C_i$, $\exists$ replica j that proposed $P'_j = P''_j$. Since the input parameter P'_j used in the third **tcast** call at replica j is guaranteed to be in the E_k set received by any replica k in the second **tcast** call, we have $C_i \subseteq E_k$, $\forall$ replicas i, k.

Hence we have $\forall$ replicas $i, j, U_i \subseteq C_j \subseteq E_i$.

Furthermore, we can derive that $\forall$ replicas i, j, if $\textbf{best}(U_i) = \textbf{best}(E_i)$, then $\textbf{best}(U_i) = \textbf{best}(C_j) = \textbf{best}(E_i)$, due to the assumption above that independent priorities never tie. □

Lemma 5.5.3 (Validity)**.** Any value decided by QuePaxa is a value proposed by some replica.

Proof. Algorithm 5 has a single decision condition: $\textbf{best}(E) = \textbf{best}(U)$. When this condition is satisfied, QuePaxa delivers $\textbf{best}(C).\textbf{value}$ as the consensus decision to the upper layer. We show that the set C contains only proposed values, hence any decided value is a proposed value.

By Lemma 5.5.2, $C \subseteq E$ in round r. We show that $\forall p \in E$, p is a value proposed by some replica. From the second **tcast** call of round r, by **tcast** property T1, there is a set S containing a majority of replicas such that $\forall j \in S$, $P_j \subseteq E$. Each replica j sets the input argument $P = P_j$ for the second **tcast** to be equal to the P set returned in the first **tcast** call of the same round r. This P set

consists of the values p received from a majority of the replicas as input to the first **tcast** call, which each replica i sets as $p = \langle \pi_i, i, v_i \rangle$. If $r = 0$, then v_i is proposed by some replica, hence the validity holds. If $r > 0$, then we can inductively show that p is derived from a value that was initially proposed by some replica in round 0. Hence, the validity property holds. □

Lemma 5.5.4 (Agreement)**.** No two replicas decide differently.

Proof. Assume by contradiction that two replicas i and j decide different values $v_i \neq v_j$ for the same slot. Without loss of generality, assume that i decides first v_i in round r_i, and j decides second the value v_j in round r_j, s.t $r_j \geq r_i$.

Case 1: $r_j = r_i$ (both replicas decide in the same round and phase).

Replica i decides $v_i = \textbf{best}(C_i).\textbf{value}$, if $\textbf{best}(U_i) = \textbf{best}(E_i)$. By Lemma 5.5.2, we have $U_i \subseteq C_i \subseteq E_i$, and we can derive that $\textbf{best}(U_i) = \textbf{best}(E_i) = \textbf{best}(C_i) = \langle \pi_i, i, v_i \rangle$. Using the same lemma for replica j's decision, we have $U_i \subseteq C_j \subseteq E_i$, and we can also derive $\textbf{best}(U_i) = \textbf{best}(E_i) = \textbf{best}(C_j) = \langle \pi_j, j, v_j \rangle$. $\textbf{best}(U_i) = \langle \pi_i, i, v_i \rangle = \langle \pi_j, j, v_j \rangle$ is a contradiction, given our assumption that i and j are different replicas and priorities are unique per proposal. Hence i and j must be the same replica and $v_i = v_j$.

Case 2: $r_j > r_i$ (replica j decides in a later round than replica i).

When replica i decides v_i, by Lemma 5.5.2 ($U_i \subseteq C_k \subseteq E_i$), any replica k advancing to round $r_i + 1$ chooses $\textbf{best}(C_k).\textbf{value} = v_i$ and proposes $\langle \pi_k, k, v_i \rangle$ in the first **tcast** of round $r_i + 1$.

Since all the sets in the three **tcast** steps are derived only from proposals containing v_i as value, we can inductively show that in any round $r_k > r_i$, any replica k chooses v_i in its proposal. Thus all the sets in any round $r > r_i$ contain only proposals with v_i as value, and thus the only decision possible is $\textbf{best}(C).\textbf{value} = v_i$. This contradicts our assumption that some replica j decides $v_j \neq v_i$ in round $r_j > r_i$.

□

Lemma 5.5.5 (Integrity)**.** No replica decides twice.

Proof. This property may be trivially satisfied by using a boolean flag decided per replica, initialized to false. A replica decides only if its decided flag is false, and sets it to true once it has decided. Due to its implementation triviality, we have not included the flag in the abstract QuePaxa algorithm. □

Lemma 5.5.6 (Liveness)**.** Each non-faulty replica eventually decides with probability 1. The expected termination time per slot is less than two asynchronous (leaderless) rounds.

Proof. Replica i decides in round r if the set U returned from i's third **tcast** call in round r contains the round's unique globally-best prioritized proposal v. Then i will necessarily choose v and deliver it.

We compute the probability that the desirable event ($v \in U$) occurs. As stated in Section 5.2.2, we assume a content-oblivious network whose choices of sets does not depend on the content of messages or their priority values. This means that replica i's U set contains more than $n/2$ proposals, each of which is equally likely to be one of the n proposed values in round r. Thus, $\Pr[v \in U] > 1/2$.

Hence we see that with probability greater than $1/2$, each replica decides in a given round. We conclude that each replica decides with probability 1 eventually, and delivers the decided value to the upper layer in less than two asynchronous rounds in expectation. □

5.5.2 Concrete QuePaxa Correctness Proofs

We prove the correctness of the concrete QuePaxa protocol (Algorithm 8 and Algorithm 6) by showing that concrete QuePaxa correctly simulates abstract QuePaxa (Algorithm 5). We first show that concrete QuePaxa computes functionally-equivalent state as the abstract QuePaxa in each step. We then show that each proposer in concrete QuePaxa eventually reaches the next step until it decides. These two statements imply that the correctness proofs for abstract QuePaxa (Section 5.5) also apply to concrete QuePaxa. Section C of QuePaxa paper [104] appendix lists the concrete QuePaxa proofs.

5.6 Implementation

We implemented QuePaxa using Go version 1.18 [72], in 4368 lines of code as counted by CLOC [97]. We use the standard Go network library and TCP [98] for reliable point-to-point links between replicas. We used Protobuf encoding [75] with the gRPC [76] plugin for remote procedure call.

Our implementation supports batching in both submitters and proposers, as well as pipelining, as in existing implementations of Rabia [122] and EPaxos [73]. The current prototype does not implement reconfiguration, but could readily be extended to do so by using consensus to agree on new configurations, as per standard existing practices [13], [14].

An open source release of our prototype is available [123].

5.6.1 Reducing the leader bottleneck in LAN scenarios

In leader-driven consensus, the leader is often a performance bottleneck because it must send n messages and receive up to n messages per commit, even on the fast path. The size of these

messages mainly depends on the batch sizes of submitters: while QuePaxa metadata is usually only a few bytes, batches of commands are often kilobytes or even megabytes.

To reduce this bottleneck in data center settings, QuePaxa leverages the property of modern data center LANs that when a node broadcasts a message m to several others, recipients usually receive m at almost the same time, within a millisecond-latency upper bound [36], [94]. After a submitter broadcasts a batch of commands to all replicas in QuePaxa, it sends only a small unique batch ID (*e.g.*, a cryptographic hash) to the consensus layer. The consensus logic then agrees on batch IDs rather than batch contents, reducing the leader's bandwidth burden. When a recorder receives a proposal containing a batch ID, it first checks whether it has received that batch's content, and if so – as is common in a data center LAN – the recorder responds immediately according to algorithm 6. If the recorder has not yet received the batch, it first requests the batch from any proposer before responding. Other protocols such as Rabia [94] and NOPaxos [36] have employed similar optimizations specific to data center networks.

5.7 Experimental evaluation

We evaluated QuePaxa to address the following key questions:

- Does QuePaxa perform comparably to state-of-the-art consensus algorithms under normal network conditions?
- Can QuePaxa offer robustness to adversarial network conditions?
- What is the impact of hedging on liveness and on recovery time?
- Can QuePaxa converge to the best hedging schedule in a realistic heterogeneous deployment?

We compare QuePaxa's performance against four state-of-the-art consensus algorithms: Multi-Paxos [13], Raft [14], Rabia [94], and EPaxos [24]. Multi-Paxos is a classic leader-based algorithm. Raft is a leader-based algorithm based on viewstamped replication [63]. Rabia uses randomization to simplify SMR, specializing in data-center networks.

EPaxos is a multi-leader protocol that partitions commands across consensus instances in parallel as their dependencies permit. EPaxos's primary goal of improving throughput via parallelism is orthogonal and complementary to QuePaxa's primary goal of robustness, and thus represents a less "apples-to-apples" baseline than Multi-Paxos and Raft, but we include it when feasible for a diverse comparison.

When feasible we use the existing Go implementations of Multi-Paxos and EPaxos [73], and of Rabia [122], each with minor enhancements for our experiments [99], [101]. We found that the existing Multi-Paxos/EPaxos code base [73] does not correctly implement leader-failure scenarios, however. Upon a leader timeout, the new leader does not initiate the prepare-promise

Algorithm	Implementation		Lines of Code	Note
Multi-Paxos	existing	[73]	2891	
EPaxos	existing	[73]	4658	
Rabia	existing	[122]	4572	VI
Multi-Paxos	new	[100]	2743	
Raft	new	[100]	2802	
QuePaxa	new	[123]	4368	

Table 5.1: Lines of code [97] in the SMR implementations.

phase. With five replicas and command execution enabled via the `-exec` flag, after any replica failure, the existing implementation never makes any progress.

Due to this issue, we use this existing EPaxos code base only for normal-case execution in Section 5.7.2 below. Other experiments use our own publicly-available Paxos and Raft implementations [100], which correctly handle replica failures.

For reference, Table 5.1 lists the SMR implementations we evaluated, each with line counts measured with CLOC [97].

5.7.1 Experimental configuration and workloads

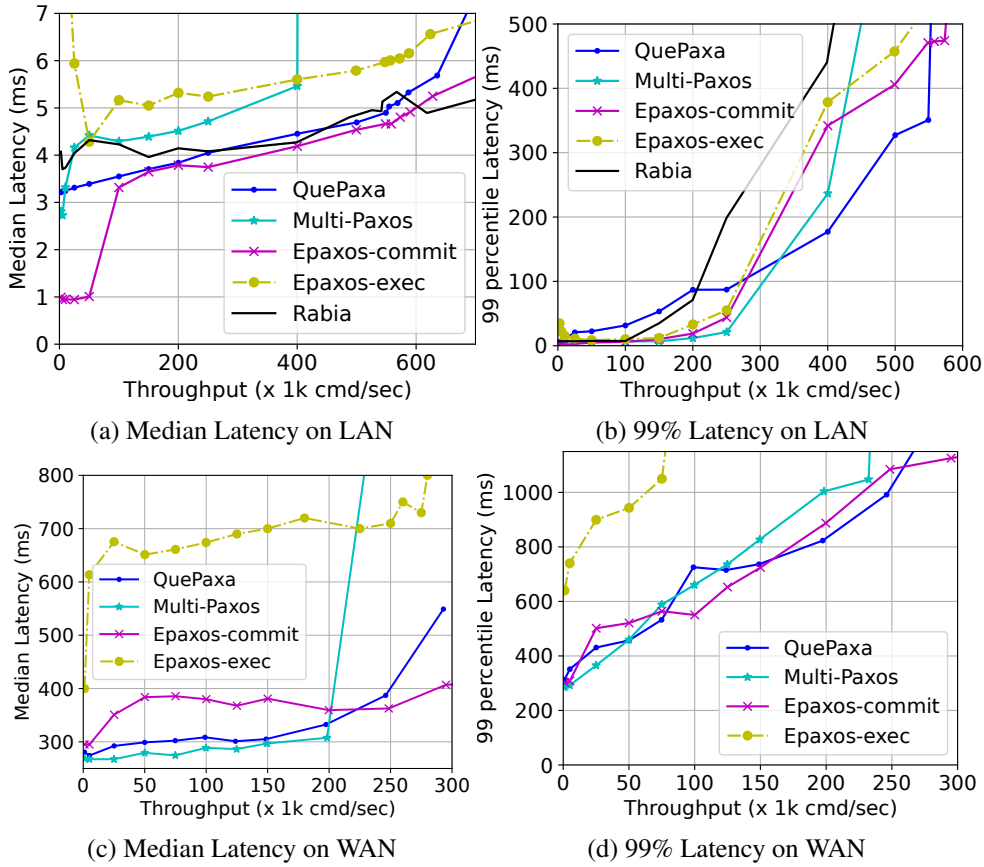
We use Amazon EC2 virtual machines [102] of type c4.4xlarge (16 virtual CPUs, 30 GB memory) and c4.2xlarge (8 virtual CPUs, 15 GB memory) for replicas and submitters, respectively. We test both a local-area-network (LAN) configuration where all replicas and submitters are located in the North Virginia AWS region, and in a wide-area-network (WAN) setup where the replicas and submitters are distributed globally across AWS regions Tokyo, Mumbai, Singapore, Ireland, and São Paulo. We use Ubuntu Linux 20.04.5 LTS [103].

Following the evaluation approach of Rabia [122], we use a string-to-string key-value store as the backend application.

Submitters generate traffic with a Poisson distribution in the open-loop model [71]. All algorithms use batching in both submitters and proposers. Multi-Paxos, EPaxos, and QuePaxa support pipelining, while the Raft and Rabia implementations do not. Client requests are 17 bytes (1-byte GET/PUT opcode plus 8-byte keys and values), consistent with request sizes common in production systems and prior research [8], [94].

For Multi-Paxos, Raft, Rabia and QuePaxa we measure end-to-end execution latency, counting the submitter-observed time required both to order and execute commands. EPaxos's execution latency is significantly higher than its commit latency [59], however, due to its dependency tracking and parallelization of commands, a feature orthogonal to this work's focus. For EPaxos, we therefore measure both ordering-with-execution latency, denoted "EPaxos-exec" in the graphs

[VI] The Rabia implementation includes log compaction; the others do not.



(a) Median Latency on LAN

(b) 99% Latency on LAN

(c) Median Latency on WAN

(d) 99% Latency on WAN

Figure 5.6: Throughput versus latency for normal-case execution, comparing QuePaxa to Rabia, Multi-Paxos and EPaxos.

below, and commit-only latency (omitting the time required to execute commands), denoted "EPaxos-commit" below.

We run each experiment for one minute, repeating experiments 3 times. We measure throughput in commands per second (cmd/sec), where a command is one 17-byte request.

5.7.2 Normal-case performance evaluation

We first evaluate the performance of QuePaxa under normal failure-free conditions, in LAN and WAN settings. We employ pipelining only in the WAN scenario (with a pipeline length of 10), as we did not observe any benefit from pipelining in the LAN case. Figure 5.6 depicts the results of this experiment.

We observe in Figure 5.6a that QuePaxa delivers LAN saturation throughput of 584k cmd/sec under a median latency upper bound of 5.8ms, compared with Multi-Paxos's saturation throughput of 400k under 5.6ms. We attribute QuePaxa's higher LAN throughput to the optimization discussed in Section 5.6.1, using submitter-driven batch dissemination to reduce critical-path

bandwidth. Multi-Paxos carries these batches in the critical path, thereby incurring higher latencies.

We see in Figure 5.6a that EPaxos-commit (without command execution) delivers LAN throughput of 699k cmd/sec under 5.8ms latency, 16.5% higher than QuePaxa's saturation throughput. This higher throughput results from EPaxos's partitioning of commands across consensus instances, a useful optimization that could be combined with QuePaxa but is outside the scope of this work. The EPaxos-commit experiment uses a conflict rate of 2%, thus committing commands in one round-trip 98% of the time. Because our QuePaxa prototype lacks this partitioning and employs only a single leader at once, its performance is naturally bottlenecked by the leader.

EPaxos-commit's 2% conflict rate impacts its 99% LAN tail latency shown in Figure 5.6b, however. In the WAN case shown in Figure 5.6c, the median latency of EPaxos-exec with command execution is 400ms higher on average than QuePaxa (in the 50k–200k cmd/sec range). This higher latency results from EPaxos's dependency management, consistent with prior observations [59], [80]. Finally, even the median WAN latency of EPaxos-commit is 60ms higher than QuePaxa in the 0-150k cmd/sec range. This is because a single conflicting command in a batch requires EPaxos to take the two-round-trip slow path, thus impacting not just tail latencies but the latencies of most commands [59].

We observe in Figure 5.6a that Rabia's median latency is comparable to QuePaxa's. As Figure 5.6b shows, however, Rabia's tail latency is 100ms-300ms higher than QuePaxa's in the throughput region 250k-400k, due to the cost of slot forfeiting in Rabia [94]. Moreover, we observed that under WAN deployment, the throughput of Rabia falls under 10 cmd/sec with latency greater than 2s. This low WAN performance stems from Rabia's assumption that network delay is small compared with the interval between consecutive requests [94, §3.2], a condition that holds in the LAN but not the WAN.

5.7.3 Scalability

This experiment evaluates the scalability of QuePaxa in a single data center (North Virginia), with an increasing number of replicas. We measure the saturation throughput of each algorithm, under a 5.8ms median latency upper bound, which we chose based on the saturation point we observed in Figure 5.6a. Figure 5.7 depicts these scalability results.

In contrast with blockchain algorithms targeting scalability up to hundreds of nodes [18], [28], crash-fault-tolerant protocols are generally deployed and evaluated at smaller scales of under 15 nodes [25], [37], so we follow this convention.

We compare QuePaxa, Multi-Paxos, and EPaxos in this experiment. We observed that EPaxos is hard-coded to support only up to 5 replicas. With more than 5 replicas, EPaxos crashes with an index-out-of-bounds exception, which stems from a hard-coded array of size 5. We reported this as a bug in the EPaxos code repository [73]. Hence, for EPaxos, we depict only the 3- and

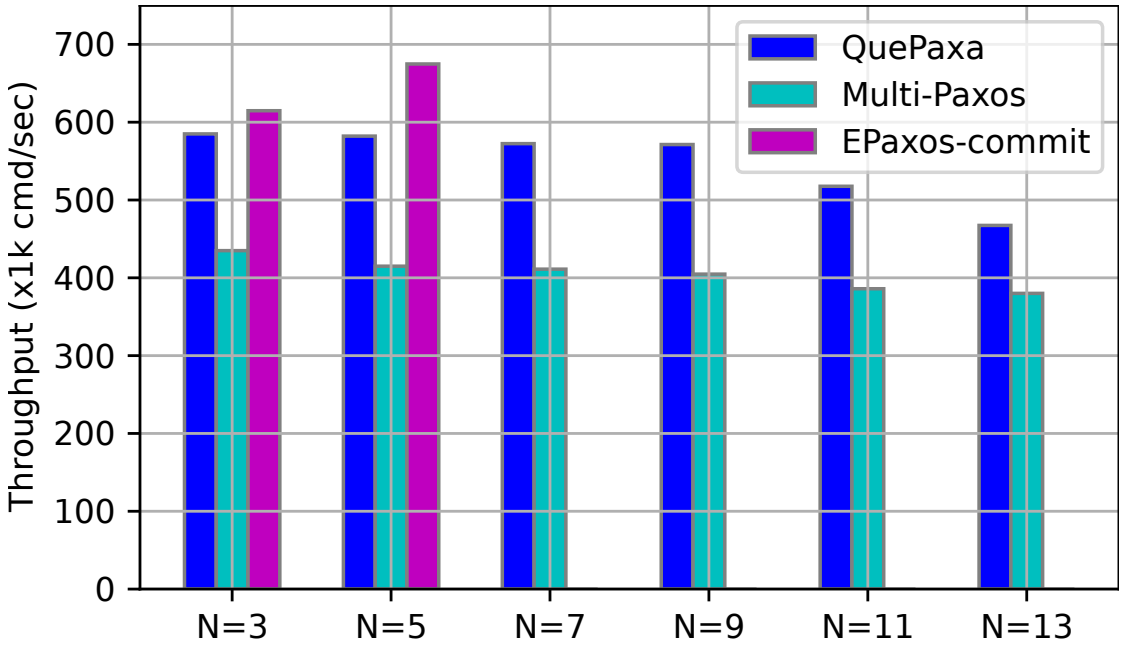


Figure 5.7: Scalability in a single data-center deployment

5-replica configurations.

We observe that the throughput of QuePaxa decreases from 584k to 467k cmd/sec as the replica count increases from 3 to 13. QuePaxa uses a quorum-based broadcast for replicating commands. When replication factor increases, the current leader in QuePaxa must exchange an increasing number of messages with non-leader replicas. This load explains the 20% throughput drop with increasing replication factor.

We observe that for all replica configuration sizes, the throughput of QuePaxa is 35% higher than Multi-Paxos, on average. We attribute this gain to QuePaxa's LAN optimization of using client replicas to disseminate the requests, reducing critical-path bandwidth usage. With this optimization disabled, we find that QuePaxa and Multi-Paxos deliver essentially the same throughput in all replica configuration sizes.

Finally, we observe that EPaxos provides better throughput than QuePaxa and Multi-Paxos. While we do not have the empirical data to showcase the scalability of EPaxos, theoretically, we expect EPaxos to scale better than QuePaxa and Multi-Paxos, because EPaxos partitions over commands and only partially orders commands. In contrast, QuePaxa and Multi-Paxos place all commands in a total order.

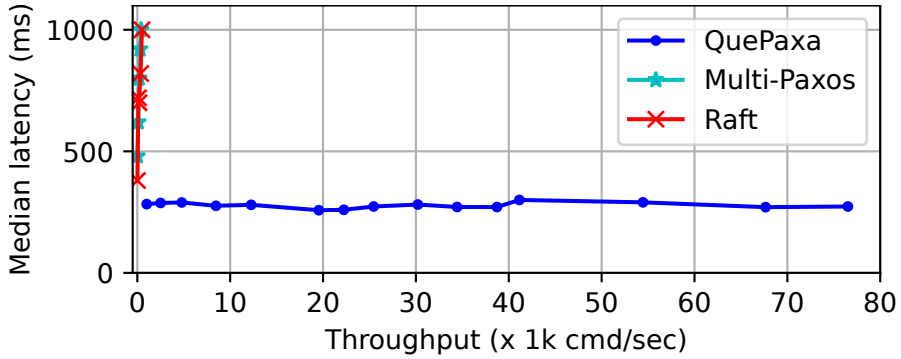


Figure 5.8: Throughput versus median latency with a network adversary that randomly attacks a minority of replicas at once.

5.7.4 Performance under adversarial network conditions

This experiment evaluates QuePaxa under simulated attack by a network adversary, similar to attacks used in recent consensus robustness work [22], [91], [124]. This adversary controls communication delays of a minority of replicas, with the aim of compromising the liveness and performance of consensus. Our simulated adversary is intended to model realistic network-based attacks, such as a denial-of-service (DoS) attack focused on a few replicas, or the use of BGP hijacking [118] to divert routes and gain direct control over delays between some replicas. Our simulated attacker increases the egress packet latency of a minority of replicas up to 500ms dynamically in 5s time epochs. This experiment runs in the WAN setting with 5 replicas. Figure 5.8 depicts these results.

We observe that under the simulated attack conditions, QuePaxa sustains a throughput of at least 75k cmd/sec under 380ms median latency. In contrast, the throughput of Multi-Paxos and Raft saturate at 2.5k cmd/sec. We interpret these results as confirming that QuePaxa's asynchronous core provides significant robustness under attack, in contrast with Multi-Paxos and Raft, which stall and make little or no progress when the current leader is under attack.

5.7.5 Impact of protocol delays on liveness and recovery

This experiment evaluates the impact of configured protocol delays – hedging delays in QuePaxa, and view-change timeouts in legacy protocols – on protocol liveness and recovery time after leader failure. We use five replicas in the WAN setting, among which we measured an average round-trip delay of 180ms. In this experiment, five submitters inject a constant aggregate load of 25k commands per second.

We first evaluate throughput under varying hedging delays (QuePaxa) or leader timeouts (existing protocols). We then investigate the protocols' recovery time after leader failure.

To measure recovery time, we "crash-stop" the leader at time $t = 15$ seconds, and measure the

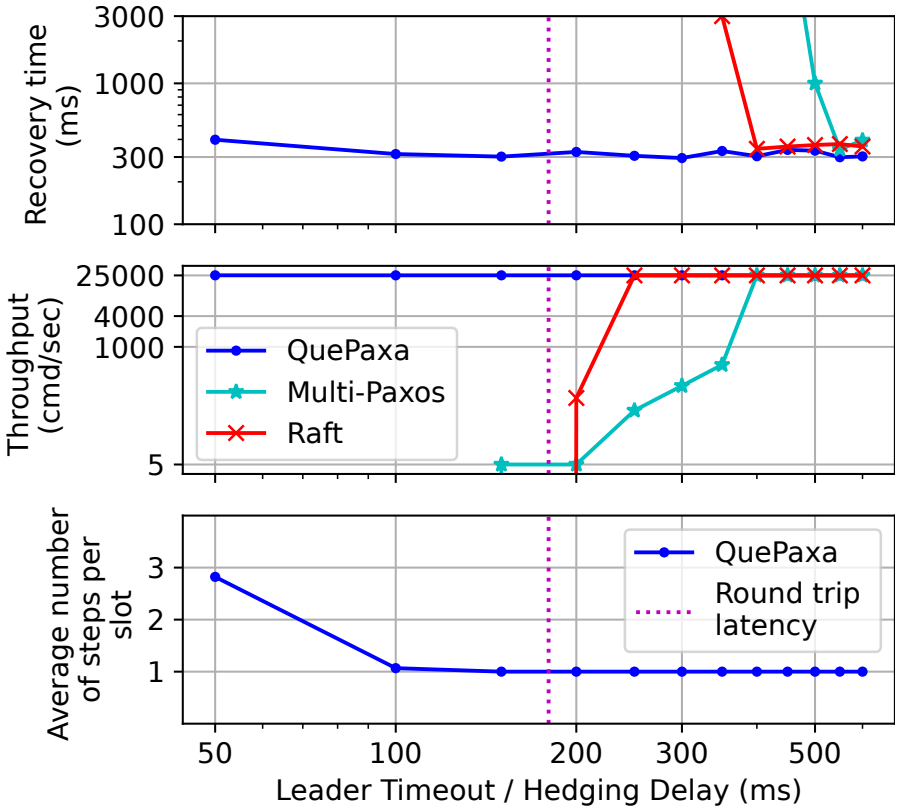


Figure 5.9: Impact of timeout/hedging delay configuration on recovery time (upper) and throughput (middle).

time it subsequently takes for a new leader or alternate proposer to resume making progress. Figure 5.9 depicts these experimental results.

Protocol Liveness: As Figure 5.9 (middle) shows, we find that QuePaxa consistently delivers throughput of 25k cmd/sec, keeping pace with the imposed load regardless of hedging delay. When QuePaxa's hedging delay is less than the average network round-trip-time of 180ms, the non-leader replicas also propose commands. However, each non-leader replica waits a short time before proposing in a slot, while the leader proposes with no delay. Even with competition from other proposers, we observe that the leader still "wins" most slots and commits in one round-trip. Moreover, when non-leader replicas propose concurrently with the leader, we find that the non-leader proposers often help the leader propagate its command, confirming that proposers effectively cooperate in QuePaxa. Even when hedging delay is less than network round-trip time, QuePaxa thus delivers stable performance.

As shown in Figure 5.9 (bottom), the main cost of hedging in QuePaxa is increased bandwidth use. When the hedging delay is less than the network round-trip time, more than one proposer submits commands, increasing message overhead.

Multi-Paxos and Raft, in contrast, rapidly lose throughput and ultimately liveness when their

view-change timeout is close to or less than the average network round-trip time. In this case, no leader can make progress without interference from false view-change triggers. With a timeout above 330ms, Multi-Paxos and Raft deliver 25k cmd/sec throughput as expected, as the leader can proceed without interruption.

We conclude that QuePaxa maintains liveness and performance with any hedging delay, while Multi-Paxos and Raft depend on their view-change timeout being correctly configured to at least about 1.8 times the network round-trip time.

Leader recovery: As Figure 5.9 (upper) shows, we see that for all hedging delays, QuePaxa exhibits a recovery time after leader failure between 303ms and 473ms. QuePaxa exhibits close to its lowest recovery time with a hedging delay around 200ms – barely above the 180ms average RTT. When hedging delay is below the RTT, recovery time increases slightly due to the redundant proposers, but nevertheless remains modest.

Multi-Paxos and Raft exhibit orders-of-magnitude higher recovery times with timeouts insufficiently above the network's RTT. Below 200ms timeouts, Multi-Paxos and Raft fail to stabilize at all and hence have no recovery time.

In Multi-Paxos, recovery time explodes with 100ms higher timeouts than Raft. This difference results from the Raft implementation using a multi-threaded gRPC design, while the Multi-Paxos code uses a single-threaded event-driven design.

With delays more than 500ms, all protocols converge to a recovery time roughly equal to network round-trip time plus the timeout, as we expect since recovery in this case depends primarily on the combination of network RTT (to run the view change) plus one timeout (to detect the need for it).

We conclude that QuePaxa robustly maintains low recovery times affected only slightly by configured hedging delay. Existing protocols effectively impose a hard lower bound on timeouts, otherwise risking high recovery time or no recovery.

5.7.6 Automatic convergence to the best leader

This experiment evaluates QuePaxa's auto-tuning mechanism to identify and converge on the best hedging schedule. We ask in particular: can QuePaxa find whichever leader maximizes performance, regardless of initial leader? This experiment uses five replicas in a single AWS region (Oregon), on five heterogeneous EC2 machines (t2.large, t2.2xlarge, c4.large, c4.xlarge, and c4.4xlarge) [102]. These EC2 types have varying computational and memory resources, with t2.large being the weakest machine and c4.4xlarge being the strongest. For each run, the t2.large machine is the initial leader. We use a constant load of 80k cmd/sec, and measure median command execution latency (Section 5.4.3). Figure 5.10 depicts these results.

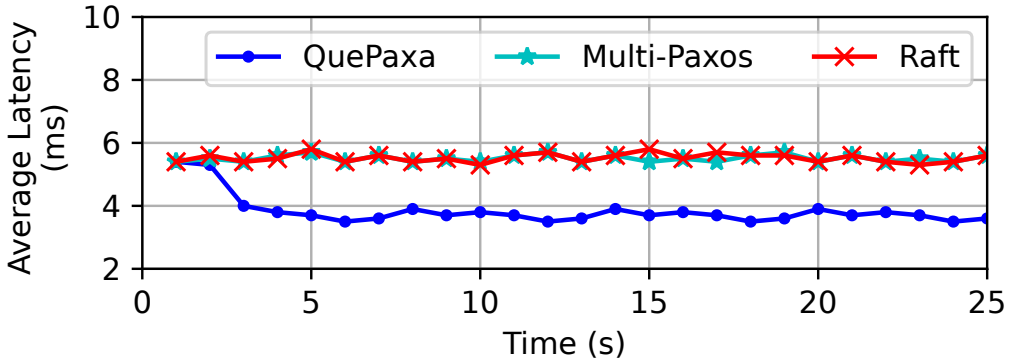


Figure 5.10: Automatic best-leader discovery in QuePaxa.

We find that Multi-Paxos and Raft maintain a high latency of 5.2 ms, retaining the slow t2.large machine as leader because it never times out. QuePaxa's multi-armed-bandit optimization, in contrast, converges to the best leader after only 4 seconds, thereafter providing a 3.8ms latency. QuePaxa thus achieves 1.4ms lower latency than Raft and Multi-Paxos in this scenario, a significant benefit in a data-center setting.

5.8 Related Work

Randomized consensus: Many algorithms use randomness to achieve asynchronous consensus [26], [113], [125]–[132]. These algorithms are rarely implemented or deployed, however, due to high complexity and poor normal-case efficiency. QuePaxa builds on ideas from QSC [113], but QuePaxa introduces a single-round-trip fast path, $O(n)$ normal-case cost, hedging, and leader-choice optimizations.

Prior hybrid consensus protocols, including the SADL-RACS in Chapter 4, have combined failure detection for synchronous performance with randomization for asynchronous robustness [21], [22], [28], [124], [133], [134]. These protocols continue to rely on timeouts to transition between synchronous and randomized modes, which adds significant complexity due to the need to manage two separate protocol stacks. Additionally, their performance is sensitive to the configuration of timeout values, often resulting in suboptimal outcomes when these values are poorly tuned. In contrast, QuePaxa employs a unified protocol stack, simplifying implementation and ensuring robust performance regardless of the chosen timeout configuration.

Rabia [94] is a randomized crash-fault-tolerant SMR scheme that uses Ben-Or's asynchronous consensus algorithm [26] as a component. Rabia specializes in low-delay, high-capacity data-center networks, however, making assumptions and design choices that limit its usefulness in other contexts. Rabia's fast path incurs three network hops and quadratic message complexity, compared with QuePaxa's two-hop, linear-complexity fast path. Rabia assumes that incoming requests are (correctly) timestamped and that "message delay is small compared to the interval between two consecutive requests" ([94, §3.2]). Experimentally, we found Rabia usable only

on low-delay LANs with high network capacity and few replicas (n=3 or 5), as examined in Section 5.7 and prior reports [124].

Hedging: Hedging is often employed in online interactive services, which typically operate under strict service-level objectives (SLOs) [110], [135]–[138]. QuePaxa is the first consensus protocol to adapt hedging to allow multiple leaders to propose, while minimizing the message overhead. Prior work has explored other ways to make consensus more robust against network performance issues, however [106], [139], [140].

Automated tuning: Most consensus protocols contain many tunable parameters: *e.g.*, leader timeout, batch size, batch time, pipeline length, garbage collection frequency. Couceiro et al. [141] used machine learning to predict the performance of total-order-broadcast protocols. Paolo et al. [142] employed multi-armed-bandit theory to tune batching in consensus protocols. QuePaxa focuses on tuning leader choice and hedging schedule, and hence is complementary to prior work. Multi-armed bandit theory has been used in many domains outside of consensus, of course [143]–[147].

Orthogonal goals: As this work focuses on the liveness and performance robustness of consensus, it does not attempt to address many other useful goals: *e.g.*, achieving scalability by partitioning over commands [24], [148] or over state [33], [149], shrinking the quorum required in the fast commit path [32], [33], exploiting WAN locality [33], [150], reducing storage costs via erasure coding [151], [152], reducing the load on leaders by outsourcing work [153], or tolerating Byzantine replica faults [18], [39]. We expect that many techniques from these complementary works could be adapted to QuePaxa, but we leave these interesting challenges to future work.

5.9 Conclusion

QuePaxa is a novel asynchronous consensus algorithm with the efficiency of partially-synchronous protocols under normal conditions, while being far more robust to challenging conditions. Our evaluation confirms that QuePaxa achieves high common-case performance, robustness to DoS attack, low recovery time, and ability to converge to the best leader.

5.9.1 Next Chapter

Chapter 3, Chapter 4, Chapter 5 addressed the robustness challenge inherent in adversarial network conditions in the setting where the nodes are non-byzantine. These three algorithms are useful in deployments where all the replicas are managed by the same trust domain; for instance an application hosted by a single cloud provider and managed by a single administrative entity.

An orthogonal setting is where a state machine replication deployment is managed by different (possibly mutually distrusting) organizations, and where up to a minority of the replicas can be Byzantine; for instance a permissioned blockchain system where up to f out of $3f+1$ nodes can

be malicious. Under such settings, none of Baxos, SADL-RACS, or QuePaxa provide safety or liveness because they assume only crash faults. In the Chapter 6 we address this by proposing *Mahi-Mahi*, a novel Byzantine fault tolerant asynchronous protocol.

6 Mahi-Mahi: Sub-second wide-area asynchronous BFT consensus

We present MAHI-MAHI, the first asynchronous BFT consensus protocol that achieves sub-second latency in a WAN setting while processing over 100,000 transactions per second. MAHI-MAHI achieves such high performance by leveraging an uncertified structured Directed Acyclic Graph (DAG) to forgo explicit certification. This reduces the number of messages required to commit and the CPU overhead for certificate verification significantly. MAHI-MAHI introduces a novel commit rule that enables committing multiple blocks in each asynchronous DAG round. MAHI-MAHI can be parametrized either with a 5 message commit delay, maximizing the commit probability under a continuously active asynchronous adversary, or with a 4 message commit delay, reducing latency under a more moderate and realistic asynchronous adversary. We demonstrate safety and liveness of MAHI-MAHI in a Byzantine context. Finally, we evaluate MAHI-MAHI in a geo-replicated setting and compare its performance against Codial-Miners and Tusk; two state-of-the-art asynchronous consensus protocols. Our evaluation shows that MAHI-MAHI achieves over 100,000 transactions per second under 1s commit latency, setting a new latency milestone in asynchronous BFT consensus protocols.

A public version of this chapter is available at https://arxiv.org/pdf/2410.08670. This work was done in collaboration with Philipp Jovanovic, Lefteris Kokoris-Kogias, Bryan Kumara, Alberto Sonnino, and Igor Zablotchi.

6.1 Introduction

The previous three chapters explored mechanisms to address the tradeoff between high performance and robustness in the context of crash fault tolerant consensus protocols. In this chapter, we shift our focus to addressing this tradeoff in the context of Byzantine fault-tolerant systems. Since a minority of nodes can deviate arbitrarily from the protocol, achieving both high performance and robustness is more challenging in the Byzantine fault-tolerant setting.

Applications that require Byzantine Fault Tolerant (BFT) consensus [154], such as blockchains [154]–[159], often rely on protocols designed for the partially synchronous network model which aim to approximate mostly benign network conditions and allow the system to perform well under these circumstances. However, protocols designed for partial synchrony lose liveness under asynchronous conditions, which can arise from poor connectivity, an active network adversary, or denial-of-service (DoS) attacks [160]. Asynchronous consensus protocols [2], [161], [162] address this issue by providing as much liveness as the network connectivity allows. To achieve this, these protocols sacrifice performance during periods of network synchrony, resulting in significantly higher latency compared to their partially synchronous counterparts. While state-of-the-art partially synchronous protocols can process over $100,000$ transactions per second with sub-second WAN latency [30], [163], current asynchronous protocols achieve similar throughput with latencies on the order of seconds [15]. This substantial latency drawback has made asynchronous consensus protocols less attractive for practical deployment.

Dual-mode protocols [16], [28] attempt to provide the best of both worlds by operating partially-synchronous consensus by default and reverting to a less performant asynchronous sub-protocol when network conditions become adverse. However, these dual-mode protocols introduce complexity and are prone to errors, as they must maintain two separate protocol stacks and implement mechanisms to detect changing network conditions and switch between the two consensus modes. Additionally, they remain vulnerable to targeted attacks that can cause the protocol to switch constantly between the two modes [16]. Due to these drawbacks, no dual-mode protocol has yet been deployed in a production environment to the best of our knowledge.

We therefore ask if it is possible to design a protocol that can simultaneously: (i) provide liveness under asynchronous network conditions, (ii) achieve latency and throughput comparable to state-of-the-art partially-synchronous consensus protocols, and (iii) maintain a simple design that allow for effective security analysis, implementation, and maintenance?

In this chapter, we introduce MAHI-MAHI, a novel low-latency and high-throughput asynchronous consensus protocol that simultaneously achieves these goals. MAHI-MAHI accomplishes this through a combination of the following techniques. (1) While state-of-the-art asynchronous protocols, such as Tusk [15], operate over a certified Directed Acyclic Graph (DAG) and attempt to commit one leader block every 9 message delays, MAHI-MAHI utilizes an *uncertified* DAG as its core data structure. This approach eliminates the overhead associated with the reliable broadcast [2], [15] of DAG vertices and allows MAHI-MAHI to commit most blocks with only

five message delays, aligning with the theoretical results of Cordial Miners [29]. (2) MAHI-MAHI introduces a novel commit rule that enables the commitment of multiple leader blocks in each DAG round while ensuring safety and liveness in the presence of an asynchronous adversary. (3) MAHI-MAHI also explores more practical network assumptions and can be parameterized to further reduce latency while maintaining liveness against a randomized adversary.

We implement MAHI-MAHI in Rust and show that it can process an impressive 350,000 transactions per second in geo-distributed environments with 50 nodes, all while keeping latency below 2 seconds. Additionally, MAHI-MAHI can process 100,000 transactions per second with latency below 1 second. This achievement sets a new record in the realm of asynchronous consensus protocols and was previously only attainable by partially synchronous protocols [30], [163], [164]. We further show that MAHI-MAHI maintains the same throughput while improving latency over recent state-of-the-art protocols, Tusk [15] and Cordial Miners [29] – for which we provide the first known implementation – achieving latency reductions of over 70% and 30%, respectively.

Contributions This chapter makes the following contributions:

- We introduce MAHI-MAHI, the first asynchronous consensus protocol capable of committing with sub-second latency while maintaining high throughput.
- We provide detailed algorithms and formal security proofs for MAHI-MAHI, demonstrating its safety and liveness under an asynchronous network model.
- We conduct a formal latency analysis of MAHI-MAHI, evaluating its commit probability under various network conditions.
- We present an implementation and evaluation of MAHI-MAHI, comparing it to other state-of-the-art protocols and demonstrating that MAHI-MAHI achieves the lowest commit latency among available asynchronous consensus protocols.

6.2 System Overview

We present an overview of MAHI-MAHI and the settings in which it operates.

6.2.1 Threat model, goals, and assumptions

We consider a message-passing system with $n = 3f + 1$ validators processing transactions using the MAHI-MAHI protocol. An adversary can adaptively corrupt up to f validators, referred to as *Byzantine*, who may deviate arbitrarily from the protocol. The remaining validators, called *honest*, follow the protocol. The adversary is computationally bounded, ensuring that standard cryptographic properties such as the security of hash functions, digital signatures, and other primitives hold. Under these assumptions, MAHI-MAHI is *safe* as no two correct validators commit different sequences of transactions. The communication network is asynchronous

and messages can be delayed arbitrarily, but messages among honest validators are eventually delivered. Given these conditions MAHI-MAHI is *live*, meaning honest validators eventually commit transactions. We provide proofs in Section 6.4.

Furthermore, we analyze MAHI-MAHI under the *random network model*, a variant of the asynchronous network model[I]. While the asynchronous model makes the worst-case assumption that the adversary has perpetually full control over the message schedule (*i.e.,* the order in which messages are received by honest validators), the random network model assumes that the message schedule is random (we give a more concrete definition in Section 6.2.3). We analyze MAHI-MAHI with parameters optimized for the random network model, representing an average-case evaluation. Our empirical results show that this parameterization generally outperforms a version of MAHI-MAHI configured for maximum resilience against an asynchronous adversary, all while maintaining safety and liveness guarantees.

MAHI-MAHI solves Byzantine Atomic Broadcast (BAB) [165], enabling validators to reach consensus on a sequence of messages necessary for State Machine Replication (SMR). According to the FLP impossibility result [44], BAB cannot be solved deterministically in an asynchronous setting. To address this, we employ a global perfect coin to introduce randomization, similar to previous work [27], [166]–[168]. This coin can be constructed using an adaptively secure threshold signature scheme [169], [170], with the distributed key setup performed under fully asynchronous conditions [171]–[175].

More formally, each validator v_k broadcasts messages by invoking $\mathsf{a\_bcast}_k(m, q)$, where m is the message and $q \in \mathbb{N}$ is a sequence number. Every validator v_i has an output $\mathsf{a\_deliver}_i(m, q, v_k)$, where m is the message, q is the sequence number, and v_k is the identity of the validator that initiated the corresponding $\mathsf{a\_bcast}_k(m, q)$. MAHI-MAHI implements a BAB protocol guaranteeing the following properties [27]:

- **Validity:** If an honest participant v_k calls $\mathsf{a\_bcast}_k(m, q)$, then every honest participant v_i eventually outputs $\mathsf{a\_deliver}_i(m, q, v_k)$, with probability 1.
- **Agreement:** If an honest participant v_i outputs $\mathsf{a\_deliver}_i(m, q, v_k)$, then every honest participant v_j eventually outputs $\mathsf{a\_deliver}_j(m, q, v_k)$ with probability 1.
- **Integrity:** For each sequence number $q \in \mathbb{N}$ and participant v_k, an honest participant v_i outputs $\mathsf{a\_deliver}_i(m, q, v_k)$ at most once, regardless of m.
- **Total Order:** If an honest participant v_i outputs $\mathsf{a\_deliver}_i(m, q, v_k)$ and $\mathsf{a\_deliver}_i(m', q', v'_k)$ where $q < q'$, all honest participants output $\mathsf{a\_deliver}_j(m, q, v_k)$ before $\mathsf{a\_deliver}_j(m', q', v'_k)$.

[I]The random network model is a novel approach introduced in a parallel work to this thesis. However, it is not a direct contribution of this thesis. Here, we include the random network model only to the extent necessary to explain Mahi-Mahi. A detailed discussion of the random network model is intentionally omitted in this chapter

6.2.2 Intuition behind the MAHI-MAHI design

MAHI-MAHI builds upon DAG-based consensus protocols that achieve high throughput by processing $O(n)$ blocks per round and fully utilizing network resources. While maintaining these throughput advantages, MAHI-MAHI focuses on reducing latency in asynchronous state machine replication. It introduces novel techniques to decrease the number of message delays required for block commitment and explores more practical network assumptions to further reduce average-case latency.

State-of-the-art asynchronous protocols, such as Tusk [15], operate over a certified DAG and try to commit one leader block every three certified rounds, requiring three message delays to certify each round. This results in at least nine message delays. To reduce latency, MAHI-MAHI operates over an uncertified DAG by forgoing the reliable broadcast [2], [15], [27] of DAG vertices, committing most blocks with only five message delays which matches the theoretical results of Cordial Miners [29]. This approach significantly reduces both bandwidth and computation cost, as validators send their blocks to every other validator only once per round, and they avoid the need to verify cryptographic certificates resulting from consistent broadcast.

This, however, creates the first challenge (**Challenge 1**): handling equivocations practically. Unlike certified DAG protocols [15], [16], [27], [176], [177], MAHI-MAHI cannot rely on certificates to prevent equivocations, necessitating the design of a novel commit rule to tolerate them. Cordial Miners [29] also face this challenge, but they address it by eventually excluding Byzantine validators that provably equivocate, which can take a long time in asynchrony.

While having five rounds between leaders provides a good probability of committing in asynchronous conditions, it also results in relatively high latency, which is not necessary for ensuring safety. We focus on addressing (**Challenge 2**): developing a commit rule that effectively reduces average-case latency without sacrificing worst-case liveness. We find that it is possible to reduce the number of rounds to four, achieving a balance between average-case latency in random network conditions and worst-case latency in the classic asynchronous model.

Even with this enhancement, committing only once every four message delays still results in significant latency variance for transactions that are not part of a committed leader block. A primary goal for MAHI-MAHI is to commit multiple blocks in each round, which would help ensure that the system's tail latency aligns more closely with the four-message delay. To achieve this, we need to address (**Challenge 3**): commit every block directly without relying on a sufficient round difference between leader blocks. If MAHI-MAHI were to adopt a traditional recursive commit rule [15], [29], which mandates that each leader block always references all previous leader blocks in their causal history, it would at best be able to commit once every four rounds. However, MAHI-MAHI recognizes that this causal reference is only necessary when there is no sufficient evidence to directly commit a block, which is not the typical case (Section 6.6). This insight indicates that the recursive commit rule used in prior research is overly conservative in its approach to skipping blocks, leading to unnecessary delays, particularly during benign node

crashes, which are immediately identifiable. To resolve this issue, we propose a new commit rule capable of promptly determining for each block whether it can be committed or discarded as soon as that decision is evident.

Section 6.3.2 presents the MAHI-MAHI commit rule that addresses these challenges. As a result, MAHI-MAHI is the first BFT consensus protocol capable of committing multiple blocks per round in the average case, while ensuring both safety and liveness in the asynchronous and random network models.

6.2.3 Structure of the MAHI-MAHI DAG

We present the structure of the MAHI-MAHI DAG, building an uncertified DAG that offers similar guarantees to a certified DAG, as shown in related work [15], [29], [30].

The MAHI-MAHI protocol operates in a sequence of logical *rounds*. In each round, every honest validator proposes a unique signed *block*, while Byzantine validators may attempt to equivocate by sending multiple blocks or none at all. During a round, validators receive transactions from users and blocks from other validators, which they refer into their proposed blocks. A block includes hash references to blocks from prior rounds, starting with their most recent block, and adds *fresh transactions* not yet included in preceding blocks. Once a block references at least $2f+1$ blocks from the previous round, the validator signs it and broadcasts it. Clients send transactions to a validator, who adds them to their blocks. If a transaction does not finalize quickly enough, the client sends it to a different validator.

Block creation and validation A block must include at least the following elements: (1) the author A of the block and their signature on the block contents; (2) a round number R; (3) a list of transactions; (4) at least $2f+1$ distinct hashes of valid blocks from the previous round $R-1$, along with potentially others from prior rounds; and (5) a share of a global perfect coin. As already mentioned, the coin can be reconstructed from any $2f+1$ shares.

A block is *valid* if: (1) the signature is valid and the author A is part of the validator set; (2) all hashes point to distinct valid blocks from previous rounds, and the sequence of past blocks includes $2f+1$ blocks from the previous round $R-1$; and (3) the share of the global perfect coin is valid[II]. Honest validators only include valid blocks into their DAG and discard invalid ones. Furthermore, honest validators only include hashes of blocks once they have downloaded their entire causal history, ensuring that they have successfully validated the block's causal history.

Rounds and waves Figure 6.1 (left) illustrates an example of a MAHI-MAHI DAG with four validators, (v_0, v_1, v_2, v_3) when parametrized to commit in 5 rounds. For the practically efficient 4-round MAHI-MAHI, the second Boost round is omitted.

[II]Each individual share of the coin can be independently verified if the coin is implemented through a threshold signature.

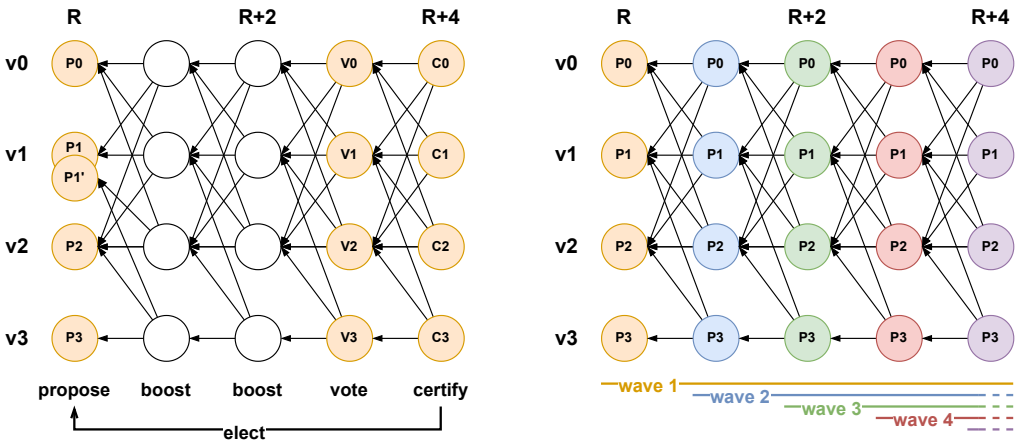


Figure 6.1: The structure of the MAHI-MAHI DAG. Left: The structure of a wave, consisting of 5 rounds (Propose, Boost, Boost, Vote, Certify). Right: Waves patterns in the MAHI-MAHI protocol (each round starts a new overlapping wave).

In its 5-rounds configuration, MAHI-MAHI defines a *wave* of 5 rounds for every block. The first round (Propose) includes the blocks that the wave attempts to commit (P_0, P_1, P_2, P_3) and the equivocating block P_1'. The second and third rounds (Boost) act as a buffer, helping to propagate these blocks to as many validators as possible. In the fourth round (Vote), every block serves as a *vote* for the first block of the Propose that it encounters when performing a depth-first search following the block hash references. In the example shown in this figure, blocks V_0, V_1, and V_2 are votes for P_0, P_1, P_2 (but not for P_1' and P_3), while block V_3 is a vote for P_0, P_1, P_2, and P_3 (but not for P_1'). The procedure $IsVote(.)$ of Algorithm 11 formally defines a vote. The fifth round (Certify) reveals which blocks from the Propose round have been implicitly certified. A block from the Propose round is considered *certified* or *has a certificate* if a block from the Certify round contains in its causal history at least $2f+1$ blocks from the Vote round that are a vote for the block. In this example, blocks C_0, C_1, C_2, and C_3 serve as certificates for P_0, P_1, and P_2. This round also opens the global perfect coin, which the decision rule (Section 6.3.2) uses to circumvent the FLP result and to commit blocks under asynchrony by electing some of the Propose blocks as *leaders*. Similar to related work [15], [27] this strategy selects leaders "after the fact" to deter a network adversary from strategically delaying leader blocks so that they are not referenced by blocks of the Vote round.

As illustrated in Figure 6.1 (right), MAHI-MAHI initiates a new wave every round. The rounds of each wave follow a consistent pattern: Propose round: R, Boost round: $R+1$, Boost round: $R+2$, Vote round: $R+3$, and Certify round: $R+4$. This pattern repeats continuously, with each new round starting a fresh wave. Algorithm 10 formally defines a wave.

Random network model We analyze MAHI-MAHI in the standard asynchronous network model, as well as in the more practical [30] random network model [15]. In the asynchronous model, the adversary chooses which blocks are received by each honest validator at each round. In contrast, the random network model assumes that at each round $R+1$, an honest validator receives and

references valid round-R blocks from a *uniformly random* subset of $2f+1$ validators. Section 6.4 provides further details and analyses the commit probability of MAHI-MAHI in both models.

6.3 The MAHI-MAHI Protocol

Algorithm 9 specifies the MAHI-MAHI main algorithm, Algorithm 10 the MAHI-MAHI decider instance, and Algorithm 11 contains various DAG helper functions. As a reminder, MAHI-MAHI operates with a single type of message: a block whose validity is described in Section 6.2.3. Validators hold these blocks in a data structure called DAG. To access the block(s) of round r authored by validator v of the DAG, we access $DAG[r, v]$. If an equivocation happened at a slot v, then $DAG[r, v]$ may return multiple blocks. To access all blocks of a given round r, we read $DAG[r, *]$.

The entry point is the procedure $ExtendCommitSequence(.)$, which is called by the application layer to extend the commit sequence. This procedure is idempotent and is called by our implementation (Section 6.5) every time the validator receives a new block. This procedure calls $TryDecide(.)$ to classify as many blocks as possible as either `commit` or `skip`. The $TryDecide(.)$ procedure iterates over all possible leaders and invokes the decider instance to classify each leader slot. The decider instance is responsible for determining the leader of a given round, certifying blocks, and classifying leader slots. The decider instance uses various helper functions, such as $IsVote(.)$, and $IsCert(.)$ of Algorithm 11), that are generic utilities for working with the DAG.

We explain MAHI-MAHI configured with a wavelength of 5 rounds. A configuration of MAHI-MAHI with a wavelength of 4 rounds operates similarly, but omits one Boost round.

6.3.1 Proposers and anchors

MAHI-MAHI leverages a perfect global coin to define several *leader slots* per round. A leader slot is a tuple (validator, round) and can be either empty or contain the validator's proposal for the respective round. If the validator is Byzantine, the slot may also contain more than one (equivocating) block. In line with related work [30], the slot can assume one of three states: `commit`, `skip`, or `undecided`. All slots are initially set to `undecided` and the goal of the protocol is to classify them as `commit` or `skip`. The number of leader slots instantiated per round and the number of boost rounds can be configured (Section 6.6 explores different configurations).

6.3.2 The MAHI-MAHI decision rule

We present the decision rule of MAHI-MAHI (see Algorithm 10 and Algorithm 11) leveraging an example protocol run. Figure 6.2 illustrates an example of a local view of a MAHI-MAHI

Algorithm 9: MAHI-MAHI Main Function

```
Data: waveLength;                                  // Set to at least 4
Data: leadersPerRound
Procedure ExtendCommitSequence(r_committed, r_highest):
    L ← TryDecide(r_committed, r_highest) // See below
    L_commit ← [] // Hold decided leader sequence
    foreach status ∈ L do
        if status = ⊥ then
            break // Stop at the first undecided leader
        end
        if status = commit(b_leader) then
            L_commit ← L_commit || b_leader
        end
    end
    return LinearizeSubDags(L_commit)
Procedure TryDecide(r_committed, r_highest):
    L ← [] // Hold decision of each leader
    for r ← r_highest to r_committed + 1 by-1 do
        for l ← leadersPerRound − 1 to 0 by −1 do
            i ← r % waveLength;
            D ← Decider(waveLength, i, l) // See Algorithm 10
            w ← D.WaveNumber(r);
            if D.ProposeRound(w) ≠ r then
                continue // Skip if not a leader
            end
            status ← D.TryDirectDecide(w) // Apply direct decision rule
            if status = ⊥ then
                status ← D.TryIndirectDecide(w) // Apply indirect
                    decision rule
            end
            L ← status || L;
        end
    end
    return L // May still contain undecided leaders
```

Algorithm 10: MahiMahi Decider Instance

```
Data: waveLength;      // Set to at least 4 (see Section 6.3)
Data: waveOffset;      // Offset creating overlapping waves (Section 6.2)
Data: leaderOffset;    // Each decider operates on a unique leader slot
Procedure WaveNumber(r):
  return (r − waveOffset)/waveLength
Procedure ProposeRound(w):
  return w · waveLength + waveOffset // See Figure 6.1
Procedure CertifyRound(w):
  return w · waveLength + waveLength − 1 + waveOffset // See Figure 6.1
Procedure VoteRound(w):
  return Self.CertifyRound(w) − 1 // See Figure 6.1
Procedure LeaderBlock(w):
  r_propose, r_certify ← Self.ProposeRound(w), Self.CertifyRound(w);
  c ← CombineCoinShares(b.share s.t. b ∈ DAG[r_certify, *]) // Common coin
  l ← c + leaderOffset // Modulo committee size
  return DAG[r_propose, l] // May return more than one block in case of equivocations
Procedure SkippedLeader(w, b_leader):
  r_vote ← Self.VoteRound(w);
  return |{¬IsVote(b, b_leader) s.t. b ∈ DAG[r_vote, *]}| ≥ 2f + 1
Procedure SupportedLeader(w, b_leader):
  r_certify ← Self.CertifyRound(w);
  return |{IsCert(b, b_leader) s.t. b ∈ DAG[r_certify, *]}| ≥ 2f + 1
Procedure TryDirectDecide(w):
  for b_leader ∈ Self.LeaderBlock(w) do
    // Loop over equivocations
    if Self.SkippedLeader(w, b_leader) then
      return skip(w)
    end
    if Self.SupportedLeader(w, b_leader) then
      return commit(b_leader)
    end
  end
  return ⊥
Procedure TryIndirectDecide(w, S):
  s_anchor ← find first s ∈ S s.t. r_certify < s.round ∧ s ≠ skip(w);
  if s_anchor = commit(b_anchor) then
    if ∃b_leader ∈ Self.LeaderBlock(w) s.t. IsCertifiedLink(b_anchor, b_leader) then
      return commit(b_leader)
    end
    else
      return skip(w)
    end
  end
  return ⊥ // The anchor is undecided or not found
```

Algorithm 11: DAG Helper Functions

Procedure `IsVote` (b_{vote}, b_{leader}) **:**
  $(id, r) \leftarrow (b_{leader}.\text{author}, b_{leader}.\text{round})$;
  return `VotedBlock`$(b_{vote}, id, r) = b_{leader}$;
Function `VotedBlock` (b, id, r) **:**
  if $r \geq b.round$ **then**
    return $\perp$
  end
  for $b' \in b.parents$ **do**
    if $(b'.author, b'.round) = (id, r)$ **then**
      return b'
    end
    $res \leftarrow$ `VotedBlock`(b', id, r);
    if $res \neq \perp$ **then**
      return res
    end
  end
  return $\perp$
return
Procedure `IsCert` (b_{cert}, b_{leader}) **:**
  $res \leftarrow |\{b \in b_{cert}.\text{parents} : \texttt{IsVote}(b, b_{leader})\}|$;
  return $res \geq 2f + 1$;
Procedure `IsLink` (b_{old}, b_{new}) **:**
  return *exists a sequence of* $k \in \mathbb{N}$ *blocks* $b_1, \ldots, b_k$ *s.t.* $b_1 = b_{old}$, $b_k = b_{new}$ *and*
   $\forall j \in [2, k] : b_j \in \bigcup_{r \geq 1} DAG[r, *] \wedge b_{j-1} \in b_j.parents$
Procedure `IsCertifiedLink` (b_{anchor}, b_{leader}) **:**
  $w \leftarrow$ WaveNumber(b_{leader}.round);
  $B \leftarrow$ GetDecisionBlocks(w);
  return $\exists b \in B$ *s.t.* `IsCert`$(b, b_{leader}) \wedge$ `IsLink`(b, b_{anchor});
Procedure `LinearizeSubDags` (L) **:**
  $O \leftarrow$ [] `// Hold output sequence`
  for $b_{leader} \in L$ **do**
    $B \leftarrow \{b \in \bigcup_{r \geq 1}$ `DAG`$[r, *]$ s.t. `IsLink`$(b, b_{leader}) \wedge b \notin O \wedge b$ not already output$\}$;
    for $b \in B$ ***in any deterministic order*** **do**
      $O \leftarrow O \,||\, b$;
    end
  end
  return O;

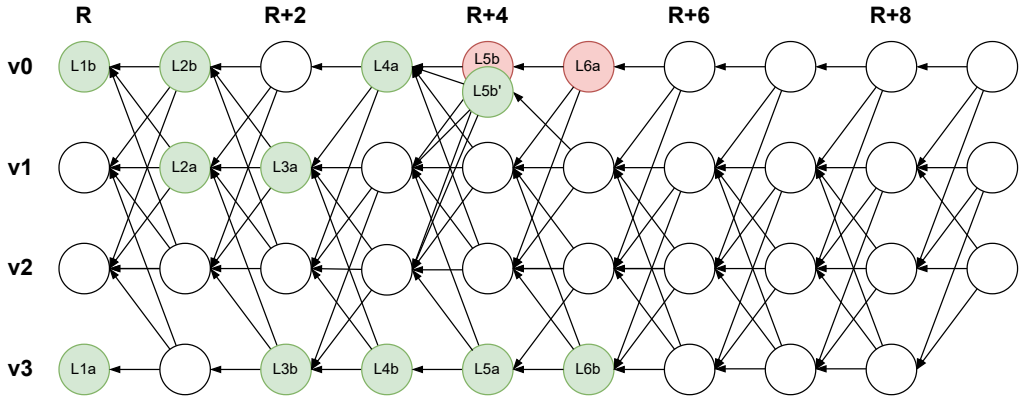


Figure 6.2: Example execution with 4 validators, wave length of 5 rounds and 2 leader slots per round.

validator, in a system with four validators, (v_0, v_1, v_2, v_3) and parameterized with two leader slots per round. In this example, we refer to blocks using the notation $B_{(v_i,R)}$, where v_i is the issuing validator and R is the block's round.

All proposer slots are initially in the `undecided` state. The validator holds the portion of the DAG depicted in Figure 6.2 and attempts to classify as many blocks in the leader slots as possible as either `commit` or `skip`.

Step 1: Determine the leader slots The validator begins by reconstructing the global perfect coin to determine the leader slots for each round. As shown in Figure 6.1 (Left), the coin shares embedded in round $R+4$[III] (the Certify round) deterministically establish the leader slots for round R (the Propose round).

In this example, the validator reconstructs the coin from any set of $2f+1$ blocks from round $R+4$ of a wave, then uses it as a seed to deterministically select two leader slots for round R: L_{1a} and L_{1b}, as illustrated in Figure 6.2. The coin also imposes an order between these two slots: by convention, L_{1a} is the *first* leader slot and L_{1b} is the *second* leader slot of round R. The validator repeats this process for every subsequent wave, determining leader slots L_{2a} and L_{2b} from the coin shares in round $R+5$, L_{3a} and L_{3b} from those in round $R+6$, and so on. The validator then sorts these leader slots in descending order: $[L_{6b}, L_{6a}, L_{5b}, L_{5a}, L_{4b}, L_{4a}, L_{3b}, L_{3a}, L_{2b}, L_{2a}, L_{1b}, L_{1a}]$.

This mechanism of determining multiple, potentially empty, leader slots from a global perfect coin is the first step towards addressing **challenge 3** (Section 6.2). Even if validators have different views of the DAG, they will still deterministically try to decide the same leader slots, in the same order, for a given round—regardless of whether they have a block for that slot in memory. This enables MAHI-MAHI to achieve low latency by electing more than one leader per round and using these slots to order the blocks they causally refer, as described below.

Step 2: Direct decision rule The validator attempts to classify each slot, even those without

[III] Or $R+3$ when MAHI-MAHI is configured with a wave length of 4 rounds.

a block as either `commit` or `skip`. To do so, the validator processes each slot individually, starting with the highest (L_{6b}), applying the MAHI-MAHI *direct decision rule*. The validator classifies a block B in a slot as `skip` if it observes $2f+1$ blocks from the subsequent Vote round that do not encounter B when performing a depth-first search following the blocks' references, and as `commit` if it observes $2f+1$ *certificates* over it. As discussed in Section 6.2, a certificate over a block B is a block from the Certify round that references at least $2f+1$ blocks from the Vote round, each of which encounter B when performing a depth-first search starting at the voting block. Otherwise, the validator leaves the slot as `undecided` (for now).

In this example, the validator targets L_{6b} first. It observes that $B_{(v_0,R+9)}$, $B_{(v_1,R+9)}$, and $B_{(v_2,R+9)}$ are certificates for L_{6b}. Therefore, it classifies L_{6b} as `commit`. Section 6.6 shows that this scenario is the most common (in the absence of an asynchronous adversary) and results in the lowest latency. The validator then targets L_{6a} and observes that $B_{(v_1,R+8)}$, $B_{(v_2,R+8)}$, and $B_{(v_3,R+8)}$ do not vote for it. Therefore, it classifies L_{6a} as `skip`. The presence of $2f+1$ blocks from the Vote round that do not vote for a block ensures that it will never be certified, and will thus never be committed by other validators with a potentially different local view of the DAG. Section 6.6 shows that this rule allows MAHI-MAHI to promptly skip (benign) crashed leaders to minimize their impact on the protocol's performance.

Malicious validators may attempt to equivocate by creating multiple blocks for the same slot, such as L_{5b} and L'_{5b} in this example. However, the direct decision rule ensures that at most one of these blocks will be classified as `commit`, while the others will be classified as `skip`. In this example, the block $B_{(v_0,R+7)}$ is a vote for L_{5b} (and not for L'_{5b}) as it is the first block of the slot encountered when performing a depth-first search starting at $B_{(v_0,R+7)}$ and recursively following all blocks in the sequence of block hashes. Conversely, $B_{(v_1,R+7)}$, $B_{(v_2,R+7)}$, and $B_{(v_3,R+7)}$ are votes for $L_{5b'}$.

This strategy addresses **challenge 1**. Even though Byzantine validators might equivocate by creating multiple blocks per slot, the causal references defined by the DAG allow the validator to interpret blocks from the Certify round as certificates for blocks from the Propose round. Coupled with the rule that honest validators author at most one block per round, this ensures that at most one block per slot receives a certificate, while all possible other equivocating blocks are skipped. In essence, MAHI-MAHI embeds the execution of a Byzantine consistent broadcast [2] into the DAG.

Step 3: Indirect decision rule In the (rare) case where the direct decision rule cannot classify a slot, the validator uses the MAHI-MAHI *indirect decision rule*. This rule looks at future slots to decide about the current one. First, it finds an *anchor*. This is the first block of the next wave (that is, the earliest slot with a round number $R' > R+4$) that is either still classified as `undecided` or already classified as `commit`. If the anchor is `undecided`, the validator marks the current slot as `undecided`. If the anchor is `commit`, the validator checks if it references at least one certificate over the current slot. If it does, the validator marks the current slot as `commit`. If it does not, the validator marks the current slot as `skip`. Section 6.4 shows the direct and indirect

decision rules are consistent, namely if one validator direct commits a block no honest validators will indirect skip it (and vice versa).

In this example, the validator fails to classify L_{1a} using the direct decision rule as there is only one certificate for L_{1a} and thus searches for its anchor. Since L_{6a} has been classified as `skip`, it cannot serve as an anchor; therefore, L_{6b} becomes the anchor for L_{1a}. Given that block $B_{(v_3,R+4)}$, which serves as a certificate for L_{1a}, is referenced in L_{6b}'s causal history, the validator classifies L_{1a} as `commit`.

This rule is the last step to solving **challenge 3**. It allows the validator to indirectly decide on a block by leveraging the earliest anchors rather than waiting for the next leader slot which may come much later. This enables MAHI-MAHI to eliminate the need for non-leader blocks between leader slots, achieving low latency by electing leader slots in every round.

Step 4: Leader slots sequence After processing all slots, the validator derives an ordered sequence of the blocks contained in the leader slots. It then iterates over this sequence, committing all slots marked as `commit` and skipping all slots marked as `skip`. This process continues until the validator encounters the first `undecided` slot. As demonstrated in Section 6.4, this commit sequence is safe, and eventually, all slots will be classified as either `commit` or `skip`.

In the example shown in Figure 6.2, the leader sequence output by the validator is [L_{1a}, L_{1b}, L_{2a}, L_{2b}, L_{3a}, L_{3b}, L_{4a}, L_{4b}, L_{5a}, L'_{5b}, L_{6b}] .

Step 5: Commit sequence Following the approach introduced by DagRider [27], the validator linearizes the blocks within the sub-DAG defined by each leader block by performing a depth-first search. If a block has already been linearized by a previous leader slot, it is not re-linearized. The validator processes each leader slot sequentially, ensuring that all blocks are included in the final commit sequence in the correct order, according to their causal dependencies. The procedure $LinearizeSubDags(.)$ of Algorithm 11 formally describes this process.

In this example, L_{1a} and L_{1b} do not define any sub-DAG (the example begins at round R) and are thus directly added to the commit sequence. Next, L_{2a} defines the sub-DAG $\{L_{1b}, B_{(v_1,R)}, B_{(v_2,R)}, L_{2a}\}$, which is linearized as [$B_{(v_1,R)}$, $B_{(v_2,R)}$, L_{2a}] since L_{1b} is already part of the commit sequence. The validator continues this process for each leader in the sequence, linearizing the sub-DAGs defined by L_{3b}, then L_{3a} and so forth following the procedure above. The final commit sequence is [L_{1a}, L_{1b}, $B_{(v_1,R)}$, $B_{(v_2,R)}$, L_{2a}, L_{2b}, $B_{(v_2,R+1)}$, L_{3a}, $B_{(v_3,R+1)}$, L_{3b}, $B_{(v_0,R+2)}$, $B_{(v_2,R+2)}$, L_{4a}, L_{4b}, $B_{(v_1,R+3)}$, $B_{(v_2,R+3)}$, L_{5a}, L'_{5b}, $B_{(v_1,R+4)}$, $B_{(v_2,R+4)}$, L_{6b}].

6.4 Security Proofs

This section proves the correctness of MAHI-MAHI, by showing that MAHI-MAHI satisfies the properties of Byzantine Atomic Broadcast (BAB) from Section 6.2. We prove the correctness of both the 4-round and 5-round versions of MAHI-MAHI. We start with results that hold

for both versions (these are mostly safety-related results) in Section 6.4.1 and continue with version-specific results in Section 6.4.2.

6.4.1 Common Proofs for w = 4 and w = 5

We start by proving the Total Order and Integrity properties of BAB. A crucial intermediate result towards these properties is that all honest validators have consistent commit sequences, i.e., the committed sequence of one honest validator is a prefix of another's, or vice-versa. This is shown in Lemma 6.4.5 and Lemma 6.4.6, which the following lemmas and observations build up to.

Lemma 6.4.1. If in round r, $2f+1$ blocks from distinct validators certify a block b, then all blocks at future rounds $r' > r$ will have a path to a certificate for b from round r.

Proof. We prove the lemma by induction on r'. The base case is $r' = r+1$. Let b' be a block at round r'. Since b' points to $2f+1$ blocks at round r, by quorum intersection, b' must point to at least one of the certificates for b.

For the induction case, assume the lemma holds up to round r' and consider the case of round $r'+1$. Let b' be a block at round $r'+1$. By the induction hypothesis, $2f+1$ blocks at round r' have paths to round-r certificates for b. Since b' points to $2f+1$ blocks from round r', by quorum intersection, b' must point to at least one block that has a path to a round-r certificate for b. □

Observation 1. *A block cannot vote for more than one block proposal from a given validator, in a given round.*

Proof. This is by construction. Honest validators interpret support in the DAG through deterministic depth-first traversal. So even if a block b in the vote round has paths to multiple leader round blocks from the same validator v (i.e., equivocating blocks), all honest validators will interpret b to vote for only one of v's blocks (the first block to appear in the depth-first traversal starting from b). □

Lemma 6.4.2. At most a single block per round from the same validator can be certified.

Proof. Assume by contradiction that in a given round r, there exist two distinct blocks b and b' from the same validator v such that both b and b' are certified. This means that there exist round-$(r+w-1)$ blocks c_b and $c_{b'}$ that certify b and b', respectively. c_b and $c_{b'}$ must point to $2f+1$ votes for b and b', respectively. By quorum intersection, there exists an honest validator that has voted for both b and b' in the vote round. Since honest validators only produce a single block per round, this implies that there exists a block that votes for both b and b', contradicting Observation 1.

□

Observation 2. *If an honest validator v directly or indirectly commits a block b, then v's local DAG contains a certificate for b.*

Proof. This follows immediately from our direct and indirect commit rules. □

Observation 3. *Honest validators agree on the sequence of leader slots.*

Proof. This follows immediately from the properties of the common coin, see Section 6.2.1. □

Lemma 6.4.3. If an honest validator v commits some block b in a slot s, then no other honest validator decides to directly skip the slot s.

Proof. Assume by contradiction that some honest validator v' decides to directly skip s. Then it must be the case that in the local DAG of v', at least $2f+1$ validators did not vote for b. However, since v commits b at s, by Observation 2, there must exist a certificate for b at s. So in v's local DAG there must be $2f+1$ validators that vote for b. By quorum intersection, at least one honest validator both voted for b and did not vote for b. Since honest validators produce a single block in the vote round, this is a contradiction. □

Lemma 6.4.4. If an honest validator directly commits some block in a slot s, then no other honest validator decides to skip the slot s.

Proof. Assume by contradiction that an honest validator v directly commits block b in slot s while another honest validator v' decides to skip s. By Lemma 6.4.3, v' cannot directly skip s; it must be the case therefore that v' skips s using the indirect decision rule. Let r be the round of s. Since v directly commits b, there exist $2f+1$ certificates for b at s. Therefore, by Lemma 6.4.1, all blocks at rounds $r' > r + w - 1$, including the anchor of s, have a path to a certificate for b at s. Thus, v' cannot decide to skip s using the indirect decision rule. We have reached a contradiction. □

Lemma 6.4.5. If a slot s is committed at two honest validators, then s contains the same block at both validators.

Proof. Let v and u be two honest validators and assume that v commits block b at slot s. We will show that if u commits slot s, then s contains b at s. Let w be the validator that produced block b. By Observation 2, for b to be committed at slot s at v, there must exist at least one certificate for b. By Observation 3, v and u agree that s must contain a block by w. By Lemma 6.4.2, at most a single block per round from w can be certified. So u cannot have a certificate for any other block than b at slot s. □

We say that a slot is *decided* at a validator v if s is committed or skipped, *i.e.,* if it is categorized as `commit` or `skip`. Otherwise, s is *undecided*.

Lemma 6.4.6. If a slot s is decided at two honest validators v and v', then either both validators commit s, or both validators skip s.

Proof. Assume by contradiction that there exists a slot s such that v and v' decide differently at s. We consider a finite execution prefix and assume *wlog* that s is the highest slot at which v and v' decide differently (*). Further assume *wlog* that v commits s and v' skips s. By Lemma 6.4.3 and Lemma 6.4.4, neither v nor v' could have used the direct decision rule for s; they must both have used the indirect rule. Consider now the anchor of s: v and v' must agree on which slot is the anchor of s, since by our assumption (*) above, they make the same decisions for all slots higher than s, including the anchor of s. Let s' be the anchor of s; s' must be committed at both v and v'. Thus, by Lemma 6.4.5, v and v' commit the same block b' at s'. But then v and v' cannot reach different decisions about slot s using the indirect decision rule. We have reached a contradiction. □

We have proven the consistency of honest validators' commit sequences: honest validators commit (or skip) the same leader blocks, in the same order. However, we are not done: we also need to prove that non-leader blocks are delivered in the same order by honest validators. We show this next.

Causal history & delivery conditions Consider an honest validator v. We call the *causal history* of a block b in v's DAG, the transitive closure of all blocks referenced by b in v's DAG, including b itself. In MAHI-MAHI, a block b is delivered by an honest validator v if (1) there exists a committed leader block l in v's DAG such that b is in l's causal history (2) all slots up to l are decided in v's DAG and (3) b has not been delivered as part of a lower slot's causal history. In this case we say b is *delivered at* slot s, or *delivered with* block l.

Lemma 6.4.7. If a block b is delivered by two honest validators v and v', then b is delivered at the same slot s, and b is delivered with the same leader block l, at both v and v'.

Proof. Let s be the slot at which b is delivered at validator v, and l the corresponding leader block in s, also at validator v. Consider now the slot s' at which b is delivered at validator v', and l' the corresponding leader block. Assume by contradiction that $s' \neq s$. If $s' < s$, then v would have also delivered b at slot s', since by Lemma 6.4.5 must commit the same leader blocks in the same slots, so v could not have delivered b again at slot s; a contradiction. Similarly, if $s < s'$, then v' would have already delivered b at slot s, since by Lemma 6.4.5 v and v' must have committed the same block in slot s; contradiction. Thus it must be that $s = s'$, and by Lemma 6.4.5, $l = l'$. □

We can now prove the main safety properties of MAHI-MAHI: Total Order and Integrity.

Theorem 13 (Total Order)**.** MAHI-MAHI *satisfies the total order property of Byzantine Atomic Broadcast.*

Proof. This property follows immediately from Lemma 6.4.7 and the fact that honest validators order the causal histories of committed blocks using the same deterministic function, and deliver blocks in this order. □

Theorem 14 (Integrity)**.** MAHI-MAHI *satisfies the integrity property of Byzantine Atomic Broadcast.*

Proof. This is by construction: a block b is delivered as part of the causal history of a committed leader block only if b has not been delivered along with an earlier leader block (see "Causal history & delivery conditions" above). So an honest validator cannot deliver the same block twice. □

We now turn to liveness properties. The following two lemmas establish that blocks broadcast by honest validators are eventually included in all honest validators' DAGs.

Lemma 6.4.8. If a block b produced by an honest validator v references some block b', then b' will eventually be included in the local DAG of every honest validator.

Proof. This is ensured by the synchronizer sub-component in each validator: if some validator w receives b from v, but does not have b' yet, w will request b' from v; since v is honest and the network links are reliable, v will eventually receive w's request, send b' to w, and w will eventually receive b'. The same is recursively true for any blocks from the causal history of b', so w will eventually receive all blocks from the causal history of b' and thus include b' in its local DAG. □

Lemma 6.4.9. If an honest validator v broadcasts a block b, then every correct validator will eventually include b in its local DAG.

Proof. Since network links are reliable, all honest validators will eventually receive b from v. By Lemma 6.4.8, all honest validators will eventually receive all of b's causal history, and so will include b in their local DAG. □

The following crucial lemma establishes that in any round r, there is at least one block b, called a common core, such that all blocks at round $r+2$ have a path to b.

Lemma 6.4.10. For any r, there is at least one block b from round r such that any valid block from round $r+2$ has a path to b.

Proof. Consider a set B of $2f+1$ blocks in round $r+1$ from honest validators. Using B, we create a table T, as follows: for blocks $b, c \in B$, let $T[b,c] = 1$ if b in $r+1$ references c in r, $T[b,c] = 0$ otherwise. By quorum intersection, any b will reference at least $f+1$ blocks in round r that are also in B, so each row of T has at least $f+1$ entries equal to 1. Thus, T has at least

$(2f+1)(f+1)$ entries equal to 1. By a counting argument, there is a block c^* in B that has a 1 entry in at least $f+1$ rows, i.e., a block from round r which is referenced by $f+1$ blocks from round $r+1$. Let P' be the set of blocks from round $r+1$ which reference c^*. Consider now any valid block b in round $r+2$; b references $2f+1$ blocks in $r+1$, so by quorum intersection b references at least one block in B. Thus, b has a path to c^*. □

6.4.2 Specific Proofs for w = 5

We continue with proofs that are specific to the liveness of the $w=5$ version of MAHI-MAHI. We show that each wave has at least $2f+1$ leader blocks that can be directly committed (Lemma 6.4.11 and Lemma 6.4.12), and thus that each wave has a nonzero probability of directly committing at least one block (Lemma 6.4.13). We then show that each slot is eventually decided directly or indirectly (Lemma 6.4.14). Finally, we show that MAHI-MAHI satisfies the Validity and Agreement properties of BAB.

As a consequence of Lemma 6.4.10, we have the following:

Lemma 6.4.11. For any r, there exists a set S of at least $2f+1$ blocks from round r such that any valid block from round $r+3$ is a vote for every block in S.

Proof. Let $r' = r+1$. By Lemma 6.4.10, there exists a block b in round $r' = r+1$ such that any valid block from round $r'+2 = r+3$ has a path to b. Now let S be the set of blocks referenced by the block b. S must contain at least $2f+1$ blocks from round r. Every block from round $r+3$ has a path to b and thus, through b, to every block in S. □

From this we can derive the following crucial lemma:

Lemma 6.4.12. For any r, there exists a set S of at least $2f+1$ blocks from round r such that every block in S has at least $2f+1$ certificates in round $r+4$.

Proof. Take S to be the set from Lemma 6.4.11. There are at least $2f+1$ blocks in $r+4$. Any block b in round $r+4$ must reference $2f+1$ blocks from round $r+3$. By Lemma 6.4.11, every block from round $r+3$ is a vote for every block in S, so b must be a certificate for every block in S. □

We denote by $\ell \leq 3f+1$ the number of leader slots per round.

Lemma 6.4.13. Fix a round r. If $\ell > f$, then an honest validator directly commits at least one slot corresponding to round r. Otherwise, the probability that an honest validator directly commits at least one slot corresponding to round r is at least $p^\star = 1 - \frac{\binom{f}{\ell}}{\binom{3f+1}{\ell}} > 0$.

Proof. By Lemma 6.4.12, at least $2f+1$ blocks from round r can be directly committed, out of a maximum of $3f+1$ blocks. When the common coin is released in round $r+4$, it selects uniformly at random ℓ round-r blocks as the ℓ slots of round r.

In the case $\ell > f$, by quorum intersection, there exists at least one slot selected by the common coin among the $2f+1$ blocks that can be directly committed.

In the case $\ell \leq f$, we can model the number of directly committed slots in round r as a hypergeometric random variable, where a success event corresponds to selecting a slot that can be directly committed. The probability of 0 successes (i.e., not committing any slots directly) is therefore at most $\frac{\binom{f}{\ell}}{\binom{3f+1}{\ell}} < 1$. □

Lemma 6.4.14. Fix a slot s. Every honest validator eventually either commits or skips s, with probability 1.

Proof. We prove the lemma by showing that the probability of s remaining undecided forever at some honest validator is 0. In order for s to remain undecided forever, s cannot be committed or skipped directly. Furthermore, s cannot be decided using the indirect rule. This means that the anchor s' of s must also remain undecided forever, and therefore the anchor s'' of s' must remain undecided forever, and so on. The probability of this occurring is at most equal to the probability of an infinite sequence of rounds with no directly committed slots, equal to $\lim_{t\to\infty}(1-p^\star)^t = 0$, where $p^\star > 0$ is the probability from Lemma 6.4.13. □

Theorem 15 (Validity)**.** MAHI-MAHI *satisfies the validity property of Byzantine Atomic Broadcast.*

Proof. Let v be an honest validator and b a block broadcast by v. We show that, with probability 1, b is eventually delivered by every honest validator. By Lemma 6.4.9, b is eventually included in the local DAG of every honest validator. So every honest validator will eventually include a reference to b in at least one of its blocks. Let r be the highest round at which some honest validator includes a reference to b in one of its blocks. By Lemma 6.4.13, with probability 1, eventually some block b' at a round $r' > r$ will be directly committed. Block b' must reference at least $2f+1$ blocks, thus at least $f+1$ blocks from honest validators. Since all validators have b in their causal histories by round r, b' must therefore have a path to b. Lemma 6.4.14 guarantees that all slots before b' are eventually decided, so b' is eventually delivered. Thus, b will be delivered at all honest validators at the latest when b' is delivered along with its causal history. □

Theorem 16 (Agreement)**.** MAHI-MAHI *satisfies the agreement property of Byzantine Atomic Broadcast.*

Proof. Let v be an honest validator and b a block delivered by v. We show that, with probability 1, b is eventually delivered by every honest validator. Let l be the leader block with which b is delivered, and s the corresponding slot. By Lemma 6.4.14, all blocks up to and including s are eventually decided by all honest validators, with probability 1. By Lemma 6.4.5, all honest validators commit l in s. Therefore, all honest validators deliver b eventually. □

6.4.3 Specific Proofs for w = 4

The total order and integrity of Mahi-Mahi configured with a wavelength of $w = 4$ are identical to those of the $w = 5$ configuration. However, the validity and agreement proofs differ slightly between the two configurations. We demonstrate that under an asynchronous network, liveness is guaranteed for w = 4 configuration. Notably, the $w = 4$ version has a smaller probability of achieving a direct commit in each wave compared to the $w = 5$ version. The proofs for w = 4 appears in [178].

6.5 Implementation

We implemented a networked, multi-core MAHI-MAHI validator in Rust by forking the Mysticeti codebase [179], consisting of about 14,000 LOC. Our implementation utilizes `tokio` [180] for asynchronous networking and employs TCP sockets for communication. We rely on `ed25519-consensus` [181] for asymmetric cryptography and `blake2` [182] for cryptographic hashing. To ensure data persistence and crash recovery, we implemented a Write-Ahead Log (WAL) tailored to the unique requirements of our consensus protocol. Furthermore, we implemented Cordial Miners [29], a state-of-the-art DAG-based asynchronous consensus protocol, using the same system components. This enabled us to to perform a comparative evaluation with MAHI-MAHI, see Section 6.6. Since the Cordial Miners paper lacks both implementation and evaluation, we believe our implementation and evaluation are additional contributions of our work. We are open-sourcing both our implementations of MAHI-MAHI and Cordial Miners, along with our orchestration tools, to ensure reproducibility of our results[IV].

6.6 Evaluation

We evaluate the throughput and latency of MAHI-MAHI through experiments conducted on Amazon Web Services (AWS), demonstrating its performance improvements over the state-of-the-art. We evaluate MAHI-MAHI with different parametrizations, with a wave length of 4 and 5 and with different numbers of leaders per round.

We compare MAHI-MAHI against Tusk [15], as an example of certified DAG-based consensus protocol, and Cordial Miners [29], as an example of an uncertified DAG-based protocol. We

[IV] https://github.com/PasinduTennage/mahi-mahi-consensus

choose these protocols because, to the best of our knowledge, Tusk has shown the highest throughput among all published and implemented asynchronous BFT protocols when evaluated in a geo-distributed environment. Cordial Miners, while lacking an implementation and evaluation, theoretically proves excellent latency bounds and is the protocol most similar to MAHI-MAHI. We also considered a performance comparison with other recent asynchronous consensus protocols, including Pace [183], Fin [184], ParBFT [185], and SQ [186], but ultimately decided against them. The reasons for this decision is that their implementations are either closed-source, only capable of handling a limited number of block proposals (leading to crashes under sustained load), or unable to operate in a WAN environment (resulting in deadlocks after a few seconds).

Our evaluation particularly aims to demonstrate the following claims:

- C1: MAHI-MAHI has similar throughput and lower latency than the baseline state-of-the-art protocols when operating in ideal conditions.
- C2: MAHI-MAHI scales well by maintaining high throughput and low latency as the number of validators increases.
- C3: MAHI-MAHI has a similar throughput to, and lower latency than, Cordial Miners, when operating in the presence of (benign) crash faults.
- C4: MAHI-MAHI latency decreases when increasing the number of leader slots per round (up to 3 leaders per round).
- C5: MAHI-MAHI parametrized with a wave length of 4 rounds has lower latency in our geo-replicated network than when configured with a wave length of 5 rounds.

Note that evaluating the performance of BFT protocols in the presence of Byzantine faults is an open research question [187], and state-of-the-art evidence relies on formal proofs of safety and liveness (presented in Section 6.4). While there is a need to robustly tolerate Byzantine faults, we note that they are rare in observed delegated proof-of-stake blockchains, as compared to crash faults which occur commonly [30].

6.6.1 Experimental Setup

We deploy MAHI-MAHI on AWS, using `m5d.8xlarge` instances across 5 different AWS regions: Ohio (us-east-2), Oregon (us-west-2), Cape Town (af-south-1), Hong Kong (ap-east-1), and Milan (eu-south-1). Validators are distributed across those regions as equally as possible. Each machine provides 10 Gbps of bandwidth, 32 virtual CPUs (16 physical cores) on a 3.1 GHz Intel Xeon Skylake 8175M, 128 GB memory, and runs Linux Ubuntu server 22.04. We select these machines because they provide decent performance and are in the price range of "commodity servers".

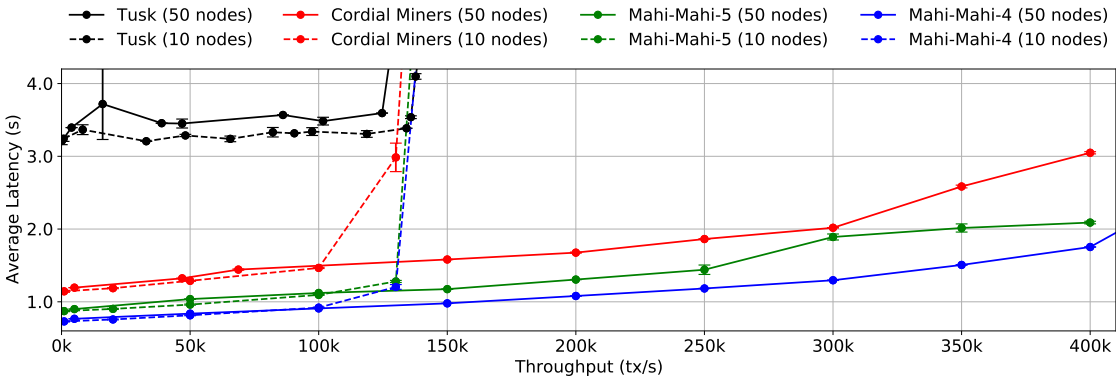


Figure 6.3: Comparative throughput-latency performance of MAHI-MAHI, Tusk, and Cordial Miners. WAN measurements with 10 and 50 validators. No validator faults. 512B transaction size.

In the following, *latency* refers to the time elapsed from the moment a client submits a transaction to when it is committed by the validators, and *throughput* refers to the number of transactions committed per second. Each data point is the average latency of 3 runs and the error bars represent one standard deviation (error bars are sometimes too small to be visible on the graph). We instantiate several geo-distributed benchmark clients within each validator submitting transactions in an open loop model, at a fixed rate. We experimentally increase the load of transactions sent to the systems, and record the throughput and latency of commits. As a result, all plots illustrate the steady-state latency of all systems under low load, as well as the maximal throughput they can provide after which latency grows quickly. Transactions in the benchmarks are arbitrary and contain 512 bytes. Unless stated otherwise, we configure MAHI-MAHI with 2 leaders per round. In the following graphs, we refer to MAHI-MAHI with a wave length of 5 as MAHI-MAHI-5 and MAHI-MAHI with a wave length of 4 as MAHI-MAHI-4.

6.6.2 Benchmark under ideal conditions

We assess the performance of MAHI-MAHI under normal, failure-free conditions in a wide-area network (WAN) environment. Figure 6.3 presents the performance results of MAHI-MAHI in a geo-replicated setting, comparing both a small committee of 10 validators and a large committee of 50 validators.

For a small committee of 10 nodes, all three systems—Tusk, Cordial Miners, and MAHI-MAHI—reach a peak throughput of approximately 100k-130k transactions per second (tx/s). However, their latencies vary significantly. Tusk and Cordial Miners achieve average latencies of 3.5s and 1.5s, respectively. In contrast, MAHI-MAHI configured with a wave length 5 has a latency of 1.1s, representing a substantial reduction of 68% compared to Tusk and 27% compared to Cordial Miners. MAHI-MAHI with wave length 4 has a latency of 0.9s, representing a substantial reduction of 74% compared to Tusk and 40% compared to Cordial Miners. Tusk's higher latency stems from its certified DAG architecture, requiring at least 9 network messages to commit a block. While Cordial Miners bypasses DAG certification, it can only commit one leader every 5

rounds. In contrast, MAHI-MAHI operating with wave length 5 consistently commits multiple blocks. MAHI-MAHI operating with wave length 4 further reduces latency as it commits blocks after 4 message delays. These results validate our claim C1.

For a large committee of 50 nodes, Figure 6.3 shows that the throughput of Cordial Miners and MAHI-MAHI exceeds 350,000 transactions per second (tx/s), while Tusk's throughput remains around 125,000 tx/s. This perhaps surprising increase in throughput occurs because our MAHI-MAHI's validator implementation is optimized for large networks and does not fully utilize all available resources (network, disk, CPU) when deployed with smaller committees. Consequently, adding more validators improves resource multiplexing, boosting MAHI-MAHI's performance. Additionally, as the committee size grows, the number of blocks per round increases, thus a larger number of blocks are included in the causal history of elected leader blocks, without incurring additional network hops. Unlike Tusk, both Cordial Miners and MAHI-MAHI experience no significant CPU overhead as the committee size increases, and bandwidth does not become a bottleneck at these throughput levels. However, we do not expect further throughput gains by increasing the committee size beyond 50 nodes (such experiments would be prohibitively expensive). As expected, Cordial Miners and MAHI-MAHI share nearly identical throughput since both rely on the same DAG implementation, and throughput is determined by the efficiency of the DAG propagation layer.

In terms of latency, Tusk and Cordial Miners achieve average latency of 3.5s and 2.6s, respectively. MAHI-MAHI parametrized with a wave length of 5 has a latency of 2s (at 350,000 tx/s), which is a 42% reduction compared to Tusk and a 23% reduction compared to Cordial Miners. MAHI-MAHI with a wave length 4 has a latency of 1.5s, which is a 57% reduction compared to Tusk and a 42% reduction compared to Cordial miners. These results validate our claim C2. Comparing the two versions of MAHI-MAHI in those two experiments also validates our claim C5.

6.6.3 Performance under faults

Figure 6.4 depicts the performance of all systems when a committee of 10 validators suffers 3 crash-faults (the maximum that can be tolerated for this committee size).

We observe that all three systems achieve a throughput of approximately $35,000$-$40,000$ tx/s. Tusk and Cordial Miners record a latency of around 7s and 1.7s, respectively. MAHI-MAHI records a latency of 0.95s and 0.85s when running with a wave length 5 and 4, respectively. Despite the presence of faulty validators, the DAG continues to collect and disseminate transactions without significant impact. The reduction in throughput seen in Figure 6.4, compared to Figure 6.3, can be attributed to two primary factors: (1) the loss of capacity due to faulty validators, and (2) the higher frequency of missing elected leader blocks, which leads to increased commit delays. MAHI-MAHI maintains a latency advantage of approximately 50% over Cordial Miners, thanks to its direct skip rule (Section 6.3), which allows MAHI-MAHI to bypass faulty leaders roughly 2 rounds earlier than Cordial Miners. Thus, our claim C3 holds.

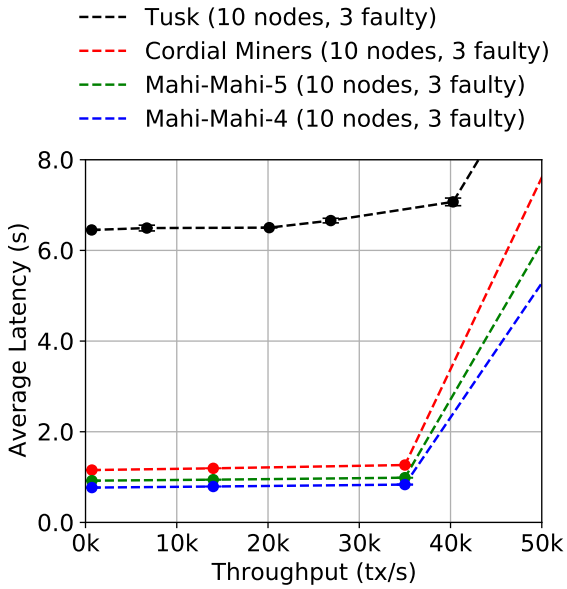
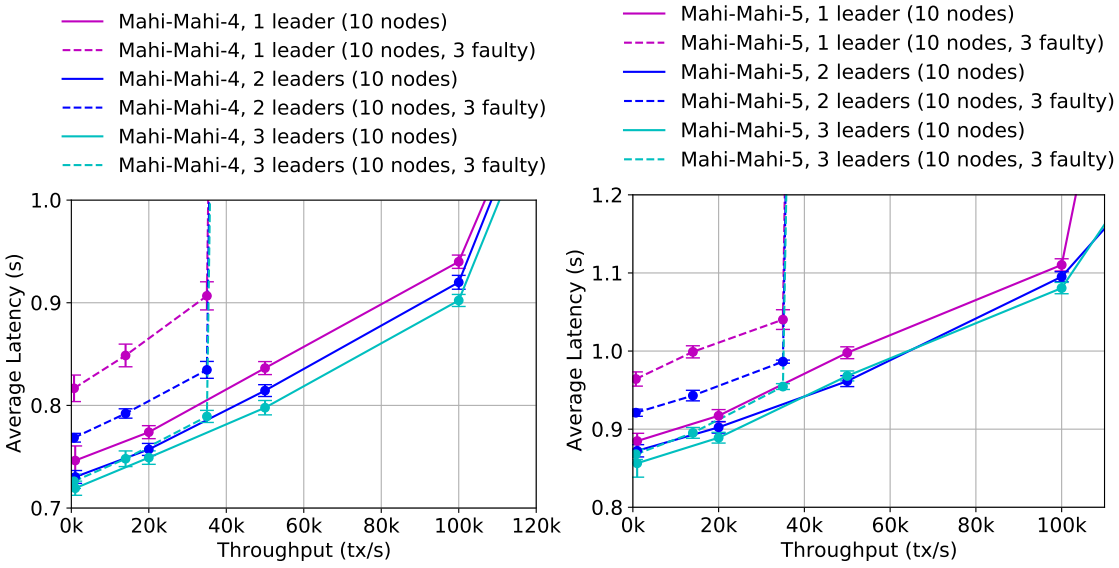


Figure 6.4: Comparative throughput-latency of MAHI-MAHI, Tusk, and Cordial Miners. WAN measurements with 10 validators. Three faults. 512B transaction size.



(a) w=4 (b) w=5

Figure 6.5: Impact of the number of leaders per round. WAN measurements with 10 validators. Zero and three faults. 512B transaction size.

6.6.4 Impact of the number of leader slots per round

Finally, we assess the impact of multiple leaders on MAHI-MAHI's performance. We experiment with MAHI-MAHI parametrized with wave lengths of 4 and 5. Figure 6.5a illustrates how MAHI-MAHI configured with a wave length of 4 rounds performs with 1, 2, and 3 leaders under both normal conditions and scenarios involving 3 crash faults. Figure 6.5b shows the same experiment with wave length 5. We observe a notable reduction in average latency as the number of leaders increases. Specifically, when the number of leaders rises from 1 to 3, MAHI-MAHI's average latency decreases by approximately 40ms in the ideal scenario, and by approximately 100ms in the crash failure scenario. This improvement arises because having more leaders per round increases the number of blocks committed directly by leaders, rather than through the causal history of previous leader blocks. These findings validate our claim C4. Increasing the number of leaders beyond 3 did not further decrease latency. This is due to the higher likelihood of failing to commit via the direct decision rule, which may cause head-of-line blocking and delays the commitment of future leaders.

6.7 Related Work

Uncertified DAG-based consensus protocols The system most similar to MAHI-MAHI is Cordial Miners [29]. Like MAHI-MAHI, Cordial Miners operates over an uncertified DAG, where each vertex represents a block that is disseminated with best-effort to all peers. The primary distinction between the two lies in their commit rules. Cordial Miners can commit at most one leader block every five rounds, which leads to significantly higher latency for transactions not included in that leader block. In contrast, MAHI-MAHI's commit rule allows for a configurable number of blocks to be committed in each round, increasing the number of blocks committed per round and reducing the latency for most transactions. MAHI-MAHI commits more blocks directly through leaders, rather than relying on the causal history of previous leader blocks. Additionally, Cordial Miners does not provide an implementation or evaluation.

Mysticeti [30] is a recent protocol that, like MAHI-MAHI, operates over an uncertified DAG but in a partially synchronous setting. Mysticeti takes advantage of synchronous periods in the network to commit blocks in three rounds, and like MAHI-MAHI, it can commit blocks every round. However, unlike MAHI-MAHI, Mysticeti completely loses liveness when the network is not synchronous. To maintain liveness in asynchronous conditions, MAHI-MAHI interprets the DAG differently from Mysticeti. Specifically, MAHI-MAHI incorporates a global perfect coin into the protocol and modifies the role of several DAG rounds to ensure that an asynchronous adversary cannot indefinitely manipulate message schedules to prevent block certificates from forming—an issue that can easily arise in Mysticeti [160].

Certified DAG-based consensus protocols DAG-Rider [27], Tusk [15], and Dumbo-NG [176] are popular asynchronous certified DAG-based consensus protocols that use reliable or consistent broadcast to explicitly certify every DAG vertex [188]. This approach introduces 3 message

delays per DAG round but simplifies the commit rule by ensuring that equivocating DAG vertices never occur. However, this method results in significantly higher latency compared to MAHI-MAHI. For instance, DAG-Rider requires at least 12 messages to commit a block, while Tusk and Dumbo-NG require 9 messages. By contrast, MAHI-MAHI can commit in just 4 or 5 message delays when respectively configured with a wave length of 4 and 5. Also, certified DAGs have higher bandwidth and CPU requirements, as validators must disseminate, receive, and verify the cryptographic certificates generated by consistent broadcast. As shown in 6.6, these factors lead to up to 70% higher latency in comparison to MAHI-MAHI.

Sailfish [189], BBCA-Chain [190], Fino [191], Shoal [164], and Shoal++[163] build on the partially synchronous version of Bullshark [192] through various improvements, including the ability to commit more blocks per round and a relaxation of DAG certification requirements. However, these protocols are limited to partially synchronous environments and, unlike MAHI-MAHI, they lose liveness in asynchronous conditions.

Linear-chain protocols Linear-chain asynchronous protocols such as Das *et al.* [193], Pace [183], FIN [184], and SQ [186] do not leverage an underlying DAG structure. They instead rely on explicit Byzantine consistent broadcast [2] and a common coin to elect a leader, whereas MAHI-MAHI incorporates these components implicitly within the DAG. Consequently, these protocols do not achieve the same level of throughput and robustness as DAG-based systems [15]. Their contributions instead lie primarily in their theoretical foundations. For example, Das *et al.* introduces a protocol that operates without a trusted setup or the need for public-key cryptography; FIN presents the first constant-time asynchronous consensus (ACS) protocol with $O(n^3)$ messages in both information-theoretic and signature-free settings; and SQ reduces this message complexity to $O(n^2)$.

Dual-mode consensus protocols Dual-mode consensus protocols, such as Ditto [28], Bullshark [16], Flexico [194], and Bolt-Dumbo [195], feature a main protocol that operates during periods of network synchrony and reverts to a backup protocol under asynchronous conditions. While this design allows for strong performance in stable network environments, dual-mode consensus protocols are inherently complex to design, prove, and implement, making them prone to errors. Consequently, to the best of our knowledge, a fully production-ready implementation of dual-mode consensus has yet to be realized[V]. Additionally, a network adversary can force constant mode switching, which severely degrades performance. This limitation arises because the protocol must wait for multiple leader timeouts in its partially synchronous mode before transitioning to its asynchronous mode. Recent advancements, such as Abraxas [197] and ParBFT [185], seek to mitigate the costs associated with mode switching by enabling the simultaneous operation of both modes. However, this approach largely increases the number of messages in the system and requires a proof of consistency across the two operational modes.

[V]The Sui team only implemented the partial-synchronous version of Bullshark [192], [196].

6.8 Conclusion

We introduce MAHI-MAHI, an asynchronous consensus protocol achieving a new performance milestone: MAHI-MAHI can process an impressive 350,000 transactions per second in geo-distributed environments with 50 nodes all while keeping latency below 2 seconds, or 100,000 transactions per second with sub-second latency—an achievement that sets a new record in the realm of asynchronous consensus protocols and that was only thought possible for partially-synchronous protocols. The exceptional performance is made possible through a novel commit rule applied over an uncertified DAG that enables commits of multiple leaders every round. This allows MAHI-MAHI to inherit the robustness and throughput inherent in DAG-based protocols, all while establishing a new standard for the latency of asynchronous consensus protocols.

7 Future Work

This thesis opens several avenues for future research.

Baxos: While Baxos achieves higher resilience against leader-based attacks, it incurs a performance overhead during normal-case executions. To address this, Baxos can be enhanced by incorporating techniques from vector clocks and causal reasoning. Instead of relying solely on random backoff, nodes could leverage causal message ordering to identify already committed instances, thereby reducing contention and avoiding unnecessary re-proposals. Preliminary studies support this hypothesis, but further detailed research and experimentation are required, which we leave as future work.

Sharded Consensus Protocols: Preliminary experiments reveal challenges when transactions require access to multiple shards, each running independent instances of state machine replication (SMR). While existing literature addresses transactions with pre-known read-write sets, providing liveness for multi-shard transaction systems where the read-write sets are not fully specified remains an open problem. For such transactions, a pre-stage is often required to discover the read-write sets, which can lead to livelocks in case of concurrent access by different transactions. Our preliminary analysis indicates that this can result in transactions with larger read-write sets losing liveness altogether, because of the inteference from the transactions with smaller read-write set.

Our initial studies suggest that a random exponential backoff (REB) mechanism can ensure eventual liveness for transactions requiring access to multiple shards, even when the full read-write set is unknwon at transaction creation. A REB mechanism, similar to Baxos, can be effectively applied in the crash fault-tolerant domain. In the Byzantine Fault Tolerant (BFT) context, integrating a verifiable random function (VRF) could enable lock-free multi-shard transactions. These preliminary findings offer a promising direction for future research and development in this area.

Optimizing SADL-RACS Latency: The current design of SADL-RACS requires four message hops per decision, resulting in higher median latency compared to Multi-Paxos. While SADL-

RACS provides higher throughput, this latency overhead can be prohibitive for applications that require low-latency responses. To address this, SADL can be interpreted as a DAG, enabling consensus to be run directly on top of SADL without incurring additional message costs.

A naive implementation of this approach would require five message hops per decision, under asynchronous network assumptions, however, our preliminary analysis suggests that this cost can be significantly reduced by implementing a two-message hop fast path and employing a hybrid commit rule. We leave the detailed design and implementation of this optimization as future work.

Auto-Tuning in QuePaxa: In QuePaxa, we employed multi-armed bandit-based auto-tuning for leader selection, opening a promising research direction for exploring tuning techniques in consensus protocols. Consensus algorithms typically have numerous tunable parameters, such as batch size, pipeline length, view timeout, default leader replica, and retry timeout, which are often manually configured in practice. As emphasized in the QuePaxa chapter, manual configuration is error-prone and can lead to suboptimal performance, especially when the operating conditions of the protocol change over time due to crashes, environmental factors, or administrative errors.

To address this, parameters can be dynamically configured using automated learning algorithms that continuously monitor the state of the consensus protocol and make necessary adjustments. Achieving this will require the application of well-established machine learning techniques, co-designed with the consensus protocols to ensure seamless integration. We leave the detailed exploration and implementation of these approaches as plausible future work.

Optimizing Mahi-Mahi: While Mahi-Mahi achieves optimal asynchronous consensus in five rounds, introducing a synchronous fast path could reduce this to three rounds under favorable network conditions. The challenge lies in safely switching between modes while maintaining termination and performance. Preliminary analysis suggests a deterministic approach, using a synchronous protocol like Mystecity [30] for the initial rounds before transitioning to Mahi-Mahi. However, further work is needed to refine this method for optimal performance.

Energy Efficient Consensus: Contributions in this thesis primarily focus on the performance and robustness of consensus protocols. However, they do not address the measurement or optimization of energy usage in such protocols. Modern cloud-based software systems are increasingly co-designed to balance high performance with energy efficiency. This opens an important avenue for future research to: (1) identify suitable metrics to quantify the energy efficiency of consensus protocols, (2) conduct rigorous analyses of energy consumption in existing consensus protocols, and (3) design new consensus protocols that minimize energy usage without compromising performance and robustness.

8 Conclusion

In this thesis, we identified five key challenges hindering the adoption of consensus algorithms in wide-area networks. To address these challenges, we proposed five novel solutions, systematically presented across four chapters. For each solution, we introduced a new protocol, provided formal proofs, developed a prototype implementation, and conducted a comprehensive evaluation, highlighting the novelty and performance of the proposed approaches.

First, we addressed the vulnerability of leader-based protocols to delayed view change attacks in wide-area networks. To resolve this, we proposed Baxos, which systematically replaces the leader election in Multi-Paxos with random exponential backoff. Our evaluation shows that Baxos improves performance by 128% compared to Multi-Paxos and Raft under delayed view change attacks. This makes Baxos a more robust solution for achieving consensus under challenging network conditions.

Second, we addressed the challenges of asynchronous network conditions and performance bottlenecks inherent in leader-based designs. To tackle these issues, we introduced SADL-RACS, a modular wide-area state machine replication protocol. SADL-RACS enhances robustness under network asynchrony and supports higher replication without compromising throughput. Our results demonstrate that SADL-RACS achieves 500,000 commands per second with an 800ms latency in wide-area setups, outperforming Multi-Paxos by 150%, effectively eliminating the leader bottleneck. Furthermore, SADL-RACS outperforms both Multi-Paxos and Raft under adversarial network conditions by ensuring liveness despite such challenges. We also showed that SADL-RACS scales effectively with the number of replicas and request sizes, making it a practical and robust solution for state machine replication.

Third, we tackled the challenges associated with protocols that rely on timeouts as a recovery mechanism. To address these issues, we introduced QuePaxa, a novel asynchronous consensus algorithm. QuePaxa combines the efficiency of partially synchronous protocols under normal conditions with exceptional robustness under adverse conditions. Our evaluation demonstrates that in the wide-area, QuePaxa achieves a comparable throughput of 250,000 requests per second to Multi-Paxos and Raft, while being significantly more resilient to challenging network

conditions. Specifically, under adversarial conditions, QuePaxa sustains 75,000 commands per second with a median latency of 380ms, whereas Multi-Paxos and Raft fail to maintain liveness. Additionally, QuePaxa exhibits faster recovery times compared to Multi-Paxos and Raft, automatically converging to the optimal leader replica through multi-armed bandit optimization, thereby ensuring superior performance.

Finally, we addressed the challenge of high latency in DAG-based Byzantine consensus protocols. To overcome this, we introduced Mahi-Mahi, a novel DAG-based asynchronous consensus protocol that employs a novel commit rule, allowing blocks to be committed without relying on an expensive certification process. Mahi-Mahi sets a new benchmark, achieving 350,000 transactions per second in geo-distributed environments with 50 nodes while maintaining sub-2-second latency. Additionally, it achieves 100,000 transactions per second with sub-second latency, establishing a new record for asynchronous consensus protocols.

We believe these contributions represent a meaningful step toward robust and high-performance wide-area consensus research. Our work spans both crash fault-tolerant and Byzantine fault-tolerant domains, ensuring its relevance to both cloud providers and public blockchains. To facilitate future research, we have made all manuscripts and codes publicly available, enabling others to build upon our findings and advance the field further.

Bibliography

[1] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem", in *Concurrency: the works of leslie lamport*, 2019, pp. 203–226.

[2] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming*. Springer Science and Business Media, 2011.

[3] J. Baker, C. Bond, J. C. Corbett, *et al.*, "Megastore: providing scalable, highly available storage for interactive services", in *Conference on Innovative Data system Research*, 2011, pp. 223–234.

[4] C. Xie, C. Su, M. Kapritsos, *et al.*, "Salt: combining {acid} and {base} in a distributed database", in *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 495–509.

[5] D. Quintero, M. Barzaghi, R. Brewster, *et al.*, *Implementing the IBM General Parallel File System (GPFS) in a Cross Platform Environment*. IBM Redbooks, 2011.

[6] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazieres, "Replication, history, and grafting in the ori file system", in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013, pp. 151–166.

[7] A. Grimshaw, M. Morgan, and A. Kalyanaraman, "Gffs—the xsede global federated file system", *Parallel Processing Letters*, vol. 23, no. 02, p. 1 340 005, 2013.

[8] N. Bronson, Z. Amsden, G. Cabrera, *et al.*, "TAO: Facebook's distributed data store for the social graph", in *USENIX Annual Technical Conference USENIX ATC 13*, San Jose, Jun. 2013, pp. 49–60.

[9] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops", in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 2011, pp. 401–416.

[10] M. Burrows, "The chubby lock service for loosely-coupled distributed systems", in *7th Symposium on Operating Systems Design and Implementation*, Seattle, 2006, pp. 335–350.

[11] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "{Zookeeper}: wait-free coordination for internet-scale systems", in *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, Boston, 2010.

[12] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: abstractions as the foundation for storage infrastructure.", in *Symposium on Operating Systems Design and Implementation OSDI*, vol. 4, 2004, pp. 8–8.

[13] L. Lamport, "Paxos made simple", *ACM SIGACT News (Distributed Computing Column) 32, 4*, vol. 32, pp. 51–58, Dec. 2001.

[14] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm", in *2014 USENIX Annual Technical Conference ATC14)*, Philadelphia, Jun. 2014, pp. 305–319.

[15] G. Danezis, L. Kokoris-Kogias, A. Sonnino, and A. Spiegelman, "Narwhal and Tusk: a DAG-based mempool and efficient BFT consensus", in *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, ACM, 2022, pp. 34–50.

[16] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, "Bullshark: DAG BFT Protocols Made Practical", in *CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[17] C. Douligeris and A. Mitrokotsa, "Ddos attacks and defense mechanisms: classification and state-of-the-art", *Computer networks*, vol. 44, no. 5, pp. 643–666, 2004.

[18] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, "HotStuff: BFT consensus with linearity and responsiveness", in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, Toronto ON Canada, Jul. 2019, pp. 347–356.

[19] T. D. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: an engineering perspective", in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, 2007, pp. 398–407.

[20] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: high-performance broadcast for primary-backup systems", in *2011 IEEE/IFIP 41st International Conference on Dependable Systems Networks (DSN)*, IEEE, 2011, pp. 245–256.

[21] S. Nikolaou and R. Van Renesse, "Turtle consensus: moving target defense for consensus", in *Proceedings of the 16th Annual Middleware Conference*, Canada, Dec. 2015, pp. 185–196.

[22] A. Spiegelman and A. Rinberg, "ACE: abstract consensus encapsulation for liveness boosting of state machine replication", *International Conference on Principles of Distributed Systems, OPODIS*, Dec. 2020.

[23] I. A. Heidi Howard, *Raft does not guarantee liveness in the face of network faults*, https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/, 2020.

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

[25] Y. Mao, F. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for WANs", in *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, Dec. 2008.

[26] M. Ben-Or, "Another advantage of free choice (extended abstract) completely asynchronous agreement protocols", in *Proceedings of the Second Annual ACM symposium on Principles of Distributed Computing*, Montreal Quebec Canada, Aug. 1983, pp. 27–30.

[27] I. Keidar, E. Kokoris-Kogias, O. Naor, and A. Spiegelman, "All You Need is DAG", in *PODC'21: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, 2021.

[28] R. Gelashvili, L. Kokoris-Kogias, A. Sonnino, A. Spiegelman, and Z. Xiang, "Jolteon and Ditto: network-adaptive efficient consensus with asynchronous fallback", in *26th International Conference on Financial Cryptography and Data Security: (FC)*, Springer, Grenada, May 2022, pp. 296–315.

[29] I. Keidar, O. Naor, O. Poupko, and E. Shapiro, "Cordial Miners: Fast and Efficient Consensus for Every Eventuality", in *37th International Symposium on Distributed Computing (DISC 2023)*, 2023.

[30] K. Babel, A. Chursin, G. Danezis, L. Kokoris-Kogias, and A. Sonnino, *Mysticeti: Low-Latency DAG Consensus with Fast Commit Path*, 2024. arXiv: 2310.14821 [cs.DC].

[31] redis, *Go lang Redis*, https://github.com/redis/go-redis, 2023.

[32] H. Howard, D. Malkhi, and A. Spiegelman, "Flexible Paxos: quorum intersection revisited", in *Proceedings of the 20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, Madrid, Spain, Dec. 2016, ISBN: 978-3-95977-031-6. DOI: 10.4230/LIPIcs.OPODIS.2016.25.

[33] A. Ailijiang, A. Charapko, M. Demirbas, and T. Kosar, "WPaxos: wide area network flexible consensus", *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 1, pp. 211–223, 2019.

[34] A. Charapko, A. Ailijiang, and M. Demirbas, "PigPaxos: devouring the communication bottlenecks in distributed consensus", in *Proceedings of the 2021 International Conference on Management of Data*, Virtual Event China, Jun. 2021, pp. 235–247.

[35] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone, "Ring Paxos: a high-throughput atomic broadcast protocol", in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, Chicago, Jun. 2010, pp. 527–536.

[36] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say NO to Paxos overhead: replacing consensus with network ordering", in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, SAVANNAH, Nov. 2016, pp. 467–483.

[37] M. Kogias and E. Bugnion, "HovercRaft: achieving scalability and fault-tolerance for microsecond-scale datacenter services", in *Proceedings of the Fifteenth European Conference on Computer Systems*, Heraklion Greece, Apr. 2020, pp. 1–17.

[38] J. C. Corbett, J. Dean, M. Epstein, *et al.*, "Spanner: google's globally distributed database", *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013.

[39] M. Castro and B. Liskov, "Practical Byzantine fault tolerance", in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleands, LA, Feb. 1999.

[40] J. Y. Halpern and X. Vilaça, "Rational consensus: extended abstract", in *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, ser. PODC '16, Chicago, Illinois, USA: Association for Computing Machinery, 2016, pp. 137–146, ISBN: 9781450339643. DOI: 10.1145/2933057.2933088. [Online]. Available: https://doi.org/10.1145/2933057.2933088.

[41] Y. Zhang, B. Shen, Y. Zhao, and A. E. Hassanien, "Rational uniform consensus with general omission failures", *Intell. Neuroscience*, vol. 2022, Jan. 2022, ISSN: 1687-5265. DOI: 10.1155/2022/9544059. [Online]. Available: https://doi.org/10.1155/2022/9544059.

[42] A. Ranchal-Pedrosa and V. Gramoli, "Rational agreement in the presence of crash faults", in *2021 IEEE International Conference on Blockchain (Blockchain)*, 2021, pp. 470–475. DOI: 10.1109/Blockchain53845.2021.00072.

[43] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony", *Journal of the ACM (JACM)*, vol. 35, no. 2, pp. 288–323, 1988.

[44] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process", *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985.

[45] S. Nakamoto, "Bitcoin: a peer-to-peer electronic cash system", 2008.

[46] J. A. Donet Donet, C. Pérez-Sola, and J. Herrera-Joancomartí, "The bitcoin p2p network", in *International conference on financial cryptography and data security*, Springer, 2014, pp. 87–102.

[47] A. Gervais, G. O. Karame, K. Wüst, V. Glykantzis, H. Ritzdorf, and S. Capkun, "On the security and performance of proof of work blockchains", in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 3–16.

[48] J. Chen and S. Micali, "Algorand", *arXiv preprint arXiv:1607.01341*, 2016.

[49] F. Saleh, "Blockchain without waste: proof-of-stake", *The Review of financial studies*, vol. 34, no. 3, pp. 1156–1190, 2021.

[50] W. Li, S. Andreina, J.-M. Bohli, and G. Karame, "Securing proof-of-stake blockchain protocols", in *Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2017 International Workshops, DPM 2017 and CBT 2017, Oslo, Norway, September 14-15, 2017, Proceedings*, Springer, 2017, pp. 297–315.

[51] A. Yakovenko, "Solana: a new architecture for a high performance blockchain v0. 8.13", *Whitepaper*, 2018.

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

[53] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai, "Sdpaxos: building efficient semi-decentralized geo-replicated state machines", in *ACM Symposium on Cloud Computing*, California, 2018, pp. 68–81.

[54] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos", in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, IEEE, 2012, pp. 1–12.

[55] J. Lockerman, J. M. Faleiro, J. Kim, *et al.*, "The fuzzylog: a partially ordered shared log", in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 357–372.

[56] L. Lamport, "Generalized consensus and Paxos", 2005.

[57] S. J. Park and J. Ousterhout, "Exploiting commutativity for practical fast replication", in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 47–64.

[58] J. Aspnes, "Randomized protocols for asynchronous consensus", *Distributed Computing*, vol. 16, no. 2-3, pp. 165–175, 2003.

[59] S. Tollman, S. J. Park, and J. K. Ousterhout, "EPaxos revisited", in *USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, Apr. 2021, pp. 613–632.

[60] T. Authors, *Tikv - distributed transactional key-value database*, https://github.com/tikv/tikv, Accessed: 2024-12-04, 2024.

[61] M. A. Bender, J. T. Fineman, S. Gilbert, and M. Young, "How to scale exponential backoff: constant throughput, polylog access attempts, and robustness", in *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SIAM, 2016, pp. 636–654.

[62] E. Ziouva and T. Antonakopoulos, "Csma/ca performance under high traffic conditions: throughput and delay analysis", *Computer communications*, vol. 25, no. 3, pp. 313–321, 2002.

[63] B. M. Oki and B. H. Liskov, "Viewstamped replication: a new primary copy method to support highly-available distributed systems", in *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, Toronto Ontario Canada, Jan. 1988, pp. 8–17.

[64] H. Howard and R. Mortier, "Paxos vs raft: have we reached consensus on distributed consensus?", in *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*, 2020, pp. 1–9.

[65] J. F. Kurose and K. W. Ross, "Computer networking", 1986.

[66] P. L. Specification, "Data over cable service interface specifications docsis 3.0", 2013.

[67] R. Van Renesse and D. Altinbuken, "Paxos made moderately complex", *ACM Computing Surveys (CSUR)*, vol. 47, no. 3, pp. 1–36, 2015.

[68] K. K. Ramakrishnan and H. Yang, "The ethernet capture effect: analysis and solution", in *Proceedings of 19th Conference on Local Computer Networks*, IEEE, 1994, pp. 228–240.

[69] L. Lamport, "The part-time parliament", 1989.

[70] E. K. Kogias, P. Jovanovic, N. Gailly, I. Khoffi, L. Gasser, and B. Ford, "Enhancing bitcoin security and performance with strong consistency via collective signing", in *25th {usenix} security symposium ({usenix} security 16)*, 2016, pp. 279–296.

[71] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open versus closed: a cautionary tale", in *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 06)*, USENIX, San Jose, CA, May 2006.

[72] J. Meyerson, "The Go programming language", *IEEE Software*, vol. 31, no. 5, pp. 104–104, 2014.

[73] I. Moraru, D. G. Andersen, and M. Kaminsky, *EPaxos go-lang*, https://github.com/efficient/epaxos/, 2013.

[74] A. Dadgar., *Raft*, https://github.com/hashicorp/raft, 2020.

[75] Google, *Protocol buffers*, https://developers.google.com/protocol-buffers/, 2020.

[76] C. N. C. Foundation, *A high performance, open source universal RPC framework*, https://grpc.io/, 2015.

[77] S. Hemminger, *NetEm – network emulator*, https://www.linux.org/docs/man8/tc-netem.html, 2011.

[78] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb", in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[79] J. Carlson, *Redis in action*. Simon and Schuster, 2013.

[80] V. S. Matte, A. Charapko, and A. Aghayev, "Scalable but wasteful: current state of replication in the cloud", in *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, Virtual USA, Jul. 2021, pp. 42–49.

[81] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion, "Energy proportionality and workload consolidation for latency-critical applications", in *Proceedings of the Sixth ACM symposium on cloud computing*, 2015, pp. 342–355.

[82] S. Micali, M. Rabin, and S. Vadhan, "Verifiable random functions", in *40th annual symposium on foundations of computer science (cat. No. 99CB37039)*, IEEE, 1999, pp. 120–130.

[83] S. Nikolaou and R. van Renesse, "Moving participants turtle consensus", *arXiv preprint arXiv:1611.03562*, 2016.

[84] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: byzantine replication under attack", *IEEE transactions on dependable and secure computing*, vol. 8, no. 4, pp. 564–577, 2010.

[85] C. Hawblitzel, J. Howell, M. Kapritsos, *et al.*, "Ironfleet: proving practical distributed systems correct", in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 1–17.

[86] E. Buchman, J. Kwon, and Z. Milosevic, "The latest gossip on bft consensus", *arXiv preprint arXiv:1807.04938*, 2018.

[87] I. Sheff, X. Wang, R. van Renesse, and A. C. Myers, "Heterogeneous paxos", in *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[88] P. Sutra, "On the correctness of egalitarian paxos", *Information Processing Letters*, vol. 156, p. 105 901, 2020.

[89] L. Lamport, "Fast paxos", *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[90] L. J. Camargos, R. M. Schmidt, and F. Pedone, "Multicoordinated paxos", in *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, 2007, pp. 316–317.

[91] P. Tennage, C. Basescu, E. K. Kogias, E. Syta, P. Jovanovic, and B. Ford, "Baxos: backing off for robust and efficient consensus", *arXiv preprint arXiv:2204.10934*, Apr. 2022.

[92] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-paxos: offloading the leader for high throughput state machine replication", in *2012 IEEE 31st Symposium on Reliable Distributed Systems*, 2012, pp. 111–120. DOI: 10.1109/SRDS.2012.66.

[93] T. Lianza and C. Snook, *Cloudflare outage*, https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/, Nov. 2020.

[94] H. Pan, J. Tuglu, N. Zhou, *et al.*, "Rabia: simplifying state-machine replication through randomization", in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, Virtual Event Germany, Oct. 2021, pp. 472–487.

[95] E. Rescorla and T. Dierks, *The transport layer security (TLS) protocol version 1.3*, RFC 8446, Aug. 2018.

[96] B. Wang, S. Liu, H. Dong, *et al.*, "Bandle: asynchronous state machine replication made efficient", in *Proceedings of the Nineteenth European Conference on Computer Systems*, ser. EuroSys '24, 2024, pp. 265–280, ISBN: 9798400704376. DOI: 10.1145/3627703.3650091.

[97] A. Danial, *Counting lines of code (CLOC)*, http://cloc.sourceforge.net/.

[98] *Transmission control protocol*, RFC 793, Sep. 1981.

[99] H. Pan, J. Tuglu, N. Zhou, *et al.*, *Rabia – modified for QuePaxa experiments*, https://github.com/dedis/quepaxa-rabia-open-loop, Sep. 2023.

[100] P. Tennage, *Paxos and Raft*, GitHub repository https://github.com/dedis/paxos-and-raft, Sep. 2023.

[101] I. Moraru, D. G. Andersen, M. Kaminsky, and P. Tennage, *EPaxos go-lang – modified for QuePaxa experiments*, https://github.com/dedis/quepaxa-ePaxos-open-loop, Sep. 2023.

[102] Amazon, *AWS instance types*, https://aws.amazon.com/ec2/instance-types/, 2023.

[103] Ubuntu, *Ubuntu Linux*, https://releases.ubuntu.com/focal/, 2023.

[104] P. Tennage, C. Basescu, L. Kokoris-Kogias, *et al.*, "QuePaxa: escaping the tyranny of timeouts in consensus", *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, Oct. 2023.

[105] M. Alimadadi, H. Mai, S. Cho, M. Ferdman, P. Milder, and S. Mu, "Waverunner: an elegant approach to hardware acceleration of state machine replication", in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, USA, 2023, pp. 357–374.

[106] H. Ng, S. Haridi, and P. Carbone, "Omni-Paxos: breaking the barriers of partial connectivity", in *Eighteenth European Conference on Computer Systems (EuroSys)*, May 2023, pp. 314–330. DOI: 10.1145/3552326.3587441.

[107] S. Rizvi, B. Wong, and S. Keshav, "Canopus: a scalable and massively parallel consensus protocol", in *Proceedings of the 13th International Conference on Emerging Networking Experiments and Technologies*, 2017, pp. 426–438.

[108] M. Whittaker, A. Ailijiang, A. Charapko, *et al.*, "Scaling replicated state machines with compartmentalization", *Proc. VLDB Endow.*, vol. 14, no. 11, pp. 2203–2215, Jul. 2021, ISSN: 2150-8097. DOI: 10.14778/3476249.3476273. [Online]. Available: https://doi.org/10.14778/3476249.3476273.

[109] C. Gorenflo, S. Lee, L. Golab, and S. Keshav, "Fastfabric: scaling hyperledger fabric to 20 000 transactions per second", *International Journal of Network Management*, vol. 30, no. 5, e2099, 2020.

[110] J. Dean and L. A. Barroso, "The tail at scale", *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[111] M. Primorac, K. J. Argyraki, and E. Bugnion, "When to hedge in interactive services", in *18th USENIX Symposium on Networked Systems Design and Implementation NSDI*, online, Apr. 2021, pp. 373–387.

[112] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous Byzantine agreement", in *ACM Symposium on Principles of Distributed Computing (PODC)*, Toronto, Ontario, Canada, Jul. 2019, pp. 337–346.

[113] B. Ford, P. Jovanovic, and E. Syta, "Que sera consensus: simple asynchronous agreement with private coins and threshold logical clocks", *arXiv preprint arXiv:2003.02291*, 2020.

[114] R. Pass and E. Shi, "Thunderella: blockchains with optimistic instant confirmation", in *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, Springer, Tel Aviv, Israel, Apr. 2018, pp. 3–33.

[115] A. Slivkins *et al.*, "Introduction to multi-armed bandits", *Foundations and Trends in Machine Learning*, vol. 12, no. 1-2, pp. 1–286, 2019.

[116] E. Gafni and L. Lamport, "Disk Paxos", in *Distributed Computing: 14th International Conference, DISC 2000*, Toledo, Spain, 2000, pp. 330–344.

[117] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of network-partitioning failures in cloud systems", in *Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, Oct. 2018.

[118] S. Cho, R. Fontugne, K. Cho, A. Dainotti, and P. Gill, "BGP hijacking classification", in *Proceedings of the Network Traffic Measurement and Analysis Conference (TMA)*, Paris, France, Jun. 2019. DOI: 10.23919/TMA.2019.8784511.

[119] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial", *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[120] B. Ford, "Threshold logical clocks for asynchronous distributed coordination and consensus", *arXiv preprint arXiv:1907.07010*, 2019.

[121] P. Whittle, "Restless bandits: activity allocation in a changing world", *Journal of Applied Probability*, vol. 25, no. A, pp. 287–298, 1988. DOI: 10.2307/3214163.

[122] H. Pan, J. Tuglu, N. Zhou, *et al.*, *Rabia*, https://github.com/haochenpan/rabia, Rabia implementation in the Go language (GitHub repository), 2021.

[123] P. Tennage, *QuePaxa*, GitHub repository https://github.com/dedis/quepaxa, Sep. 2023.

[124] P. Tennage, A. Desjardins, and E. K. Kogias, "Mandator and Sporades: robust wide-area consensus with efficient request dissemination", *arXiv preprint arXiv:2209.06152*, 2022.

[125] P. Ezhilchelvan, A. Mostefaoui, and M. Raynal, "Randomized multivalued consensus", in *Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISORC 2001*, IEEE, Magdeburg, Germany, May 2001, pp. 195–200.

[126] R. Friedman, A. Mostefaoui, and M. Raynal, "Simple and efficient oracle-based consensus protocols for asynchronous Byzantine systems", *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 1, pp. 46–56, 2005.

[127] J. Zhang and W. Chen, "Bounded cost algorithms for multivalued consensus using binary consensus instances", *Information Processing Letters*, vol. 109, no. 17, pp. 1005–1009, 2009.

[128] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous Byzantine consensus with $t < n/3$ and $O(n^2)$ messages", in *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, Paris France, Jul. 2014, pp. 2–9.

[129] M. Raynal, *Fault-tolerant Message-Passing Distributed Systems: an Algorithmic Approach*. Springer, 2018.

[130] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous Byzantine agreement with expected $\mathcal{O}(1)$ rounds, expected communication, and optimal resilience", in *Financial Cryptography and Data Security (FC)*, Springer, Frigate Bay, Feb. 2019, pp. 320–334.

[131] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 31–42.

[132] A. Oliveira, H. Moniz, and R. Rodrigues, "Alea-BFT: practical asynchronous Byzantine fault tolerance", *arXiv preprint arXiv:2202.02071*, Feb. 2022.

[133] M. K. Aguilera and S. Toueg, "Failure detection and randomization: a hybrid approach to solve consensus", *SIAM Journal of Computing*, vol. 28, no. 3, pp. 890–903, 1998. DOI: 10.1137/S0097539796312915.

[134] A. Spiegelman, "In search for an optimal authenticated Byzantine agreement", in *Proceedings of the 35th International Symposium on Distributed Computing (DISC)*, Freiburg, Germany, Oct. 2021. DOI: 10.4230/LIPIcs.DISC.2021.38.

[135] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: the google cluster architecture", *IEEE micro*, vol. 23, no. 2, pp. 22–28, 2003.

[136] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: an introduction to the design of warehouse-scale machines", *Synthesis lectures on computer architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[137] K. Gardner, S. Zbarsky, S. Doroudi, M. Harchol-Balter, and E. Hyytia, "Reducing latency via redundant requests: exact analysis", *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1, pp. 347–360, 2015.

[138] S. Pan, T. Stavrinos, Y. Zhang, *et al.*, "Facebook's tectonic filesystem: efficiency from exascale", in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 217–231.

[139] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish, "Detecting failures in distributed systems with the Falcon spy network", in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Cascais, Portugal: Association for Computing Machinery, 2011, pp. 279–294, ISBN: 9781450309776. DOI: 10.1145/2043556.2043583. [Online]. Available: https://doi.org/10.1145/2043556.2043583.

[140] K. Ngo, S. Sen, and W. Lloyd, "Tolerating slowdowns in replicated state machines using copilots", in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2020.

[141] M. Couceiro, P. Romano, and L. Rodrigues, "A machine learning approach to performance prediction of total order broadcast protocols", in *2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, IEEE, Budapest, Sep. 2010, pp. 184–193.

[142] P. Romano and M. Leonetti, "Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning", in *2012 International Conference on Computing, Networking and Communications (ICNC)*, IEEE, Hawaii, Jan. 2012, pp. 786–792.

[143] M. Abundo, V. Di Valerio, V. Cardellini, and F. L. Presti, "Bidding strategies in QoS-Aware cloud systems based on N-armed bandit problems", in *2014 IEEE 3rd Symposium on Network Cloud Computing and Applications (ncca 2014)*, IEEE, Feb. 2014, pp. 38–45.

[144] P. Dai, Z. Hang, K. Liu, *et al.*, "Multi-armed bandit learning for computation-intensive services in MEC-empowered vehicular networks", *IEEE Transactions on Vehicular Technology*, vol. 69, no. 7, pp. 7821–7834, 2020.

[145] F. Li, D. Yu, H. Yang, J. Yu, H. Karl, and X. Cheng, "Multi-armed-bandit-based spectrum scheduling algorithms in wireless networks: a survey", *IEEE Wireless Communications*, vol. 27, no. 1, pp. 24–30, 2020.

[146] J. Lu, L. Li, D. Shen, *et al.*, "Dynamic multi-arm bandit game based multi-agents spectrum sharing strategy design", in *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*, IEEE, St. Petersburg, Florida, Sep. 2017, pp. 1–6.

[147] W. Xia, T. Q. Quek, K. Guo, W. Wen, H. H. Yang, and H. Zhu, "Multi-armed bandit-based client scheduling for federated learning", *IEEE Transactions on Wireless Communications*, vol. 19, no. 11, pp. 7108–7123, 2020.

[148] V. Enes, C. Baquero, T. F. Rezende, A. Gotsman, M. Perrin, and P. Sutra, "State-machine replication for Planet-Scale systems", in *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*, Heraklion, Greece, Apr. 2020, ISBN: 978-1-4503-6882-7. DOI: 10.1145/3342195.3387543.

[149] S. Peluso, A. Turcu, R. Palmieri, G. Losa, and B. Ravindran, "Making fast consensus generally faster", in *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Toulouse, France, Jun. 2016.

[150] F. Nawab, D. Agrawal, and A. E. Abbadi, "DPaxos: managing data closer to users for low-latency and mobile applications", in *ACM SIGMOD/PODS Conference on Management of Data*, Houston, TX, USA, Jun. 2018.

[151] M. Uluyol, A. Huang, A. Goel, M. Chowdhury, and H. V. Madhyastha, "Near-optimal latency versus cost tradeoffs in geo-distributed storage", in *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*, Santa Clara, CA, USA, Feb. 2020, ISBN: 978-1-939133-13-7.

[152] Z. Wang, T. Li, H. Wang, *et al.*, "CRaft: an Erasure-coding-supported version of Raft for reducing storage cost and network cost", in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, USA, Feb. 2020.

[153] Z. Xu, C. Stewart, and J. Huang, "Elastic, geo-distributed RAFT", in *Proceedings of the International Symposium on Quality of Service*, Phoenix, Arizona: Association for Computing Machinery, 2019, ISBN: 9781450367783. DOI: 10.1145/3326285.3329046. [Online]. Available: https://doi.org/10.1145/3326285.3329046.

[154] S. Bano, A. Sonnino, M. Al-Bassam, *et al.*, *Consensus in the age of blockchains*, 2017. arXiv: 1711.03936 [cs.CR].

[155] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis, "Chainspace: A sharded smart contracts platform", *arXiv preprint arXiv:1708.03778*, 2017.

[156] M. Baudet, G. Danezis, and A. Sonnino, "Fastpay: High-performance byzantine fault tolerant settlement", in *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, 2020, pp. 163–177.

[157] S. Blackshear, A. Chursin, G. Danezis, *et al.*, "Sui lutris: A blockchain combining broadcast and consensus", *arXiv preprint arXiv:2310.18042*, 2023.

[158] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford, "OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding", in *IEEE S and P'18: Proceedings of the 39th IEEE Symposium on Security and Privacy*, IEEE, 2018, pp. 19–34.

[159] A. Sonnino, S. Bano, M. Al-Bassam, and G. Danezis, "Replay attacks and defenses against cross-shard consensus in sharded distributed ledgers", in *2020 IEEE European Symposium on Security and Privacy (Euro S and P)*, IEEE, 2020, pp. 294–308.

[160] G. Giuliari, A. Sonnino, M. Frei, F. Streun, L. Kokoris-Kogias, and A. Perrig, "An Empirical Study of Consensus Protocols' DoS Resilience", in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1345–1360.

[161] J. Chen, A. Sonnino, L. Kokoris-Kogias, and M. Sadoghi, *Thunderbolt: Causal Concurrent Consensus and Execution*, 2024. arXiv: 2407.09409 [cs.DC].

[162] Z. Ren, K. Cong, J. Pouwelse, and Z. Erkin, *Implicit Consensus: Blockchain with Unbounded Throughput*, 2017. arXiv: 1705.11046 [cs.DC].

[163] B. Arun, Z. Li, F. Suri-Payer, S. Das, and A. Spiegelman, *Shoal++: High Throughput DAG BFT Can Be Fast!*, 2024. arXiV: 2405.20488 (cs.DC).

[164] A. Spiegelman, B. Arun, R. Gelashvili, and Z. Li, "Shoal: Improving dag-bft latency and robustness", in *Financial Cryptography and Data Security: 28th International Conference, FC 2024*, 2024.

[165] F. Cristian, H. Aghili, R. Strong, and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement", *Information and Computation, Volume 118, Issue 1*, vol. 118, 1 1995.

[166] E. Blum, J. Katz, C.-D. Liu-Zhang, and J. Loss, "Asynchronous byzantine agreement with subquadratic communication", in *Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18*, Springer, 2020, pp. 353–380.

[167] C. Cachin, K. Kursawe, and V. Shoup, "Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography", in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, 2000, pp. 123–132.

[168] J. Loss and T. Moran, *Combining asynchronous and synchronous byzantine agreement: The best of both worlds*, 2018. IACR: 2018/235.

[169] R. Bacho and J. Loss, "On the Adaptive Security of the Threshold BLS Signature Scheme", in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[170] D. Boneh, B. Lynn, and H. Shacham, "Short signatures from the Weil pairing", in *International conference on the theory and application of cryptology and information security*, Springer, 2001, pp. 514–532.

[171] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, and G. Stern, "Bingo: Adaptivity and Asynchrony in Verifiable Secret Sharing and Distributed Key Generation", in *Advances in Cryptology – CRYPTO 2023*, 2023.

[172] I. Abraham, P. Jovanovic, M. Maller, S. Meiklejohn, G. Stern, and A. Tomescu, "Reaching Consensus for Asynchronous Distributed Key Generation", *Distributed Computing*, vol. 36, 3 2023.

[173] S. Das, Z. Xiang, L. Kokoris-Kogias, and L. Ren, "Practical Asynchronous High-threshold Distributed Key Generation and Distributed Polynomial Sampling", in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5359–5376.

[174] S. Das, T. Yurek, Z. Xiang, A. Miller, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous distributed key generation", in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 2518–2534. DOI: 10.1109/SP46214.2022.9833584.

[175] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman, "Asynchronous Distributed Key Generation for Computationally-Secure Randomness, Consensus, and Threshold Signatures.", in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1751–1767.

[176] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency", in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1187–1201.

[177] L. Yang, S. J. Park, M. Alizadeh, S. Kannan, and D. Tse, "{DispersedLedger}:{High-Throughput} Byzantine Consensus on Variable Bandwidth Networks", in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 493–512.

[178] P. Jovanovic, L. K. Kogias, B. Kumara, A. Sonnino, P. Tennage, and I. Zablotchi, "Mahi-mahi: low-latency asynchronous bft dag-based consensus", *arXiv preprint arXiv:2410.08670*, 2024.

[179] M. Labs, *Mysticeti: low-latency dag consensus with fast commit path*, https://github.com/asonnino/mysticeti, 2024.

[180] T. T. Team, *Tokio*, https://tokio.rs, 2024.

[181] H. de Valence, *Ed25519 for consensus-critical contexts*, https://crates.io/crates/ed25519-consensus, 2024.

[182] RustCrypto, *Rustcrypto: hashes*, https://github.com/RustCrypto/hashes, 2024.

[183] H. Zhang and S. Duan, "Pace: Fully parallelizable bft from reproposable byzantine agreement", in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 3151–3164.

[184] S. Duan, X. Wang, and H. Zhang, "FIN: Practical Signature-Free Asynchronous Common Subset in Constant Time", in *CCS '23: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 815–829.

[185] X. Dai, B. Zhang, H. Jin, and L. Ren, "ParBFT: Faster Asynchronous BFT Consensus with a Parallel Optimistic Path", in *CCS '23: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[186] X. Sui, X. Wang, and S. Duan, *Signature-Free Atomic Broadcast with Optimal O(n2) Messages and O(n1) Expected Time*, 2023. IACR: 2023/1549 (cs.DC).

[187] S. Bano, A. Sonnino, A. Chursin, *et al.*, "Twins: Bft systems made robust", in *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*, 2021.

[188] M. Raikwar, N. Polyanskii, and S. Müller, "SoK: DAG-based Consensus Protocols", in *2024 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, IEEE, 2024, pp. 1–18.

[189] N. Shrestha, R. Shrothrium, A. Kate, and K. Nayak, *Sailfish: towards improving latency of dag-based bft*, Cryptology ePrint Archive, Paper 2024/472, 2024.

[190] D. Malkhi, C. Stathakopoulou, and M. Yin, "BBCA-CHAIN: One-Message, Low Latency BFT Consensus on a DAG", in *Financial Cryptography and Data Security 2024*, International Financial Cryptography Association, 2024, pp. 1–18.

[191] D. Malkhi and P. Szalachowski, "Maximal extractable value (mev) protection on a dag", in *4th International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2022)*, Tokenomics, 2022, pp. 1–18.

[192] A. Spiegelman, N. Giridharan, A. Sonnino, and L. Kokoris-Kogias, *Bullshark: the partially synchronous version*, arXiv preprint arXiv:2209.05633, 2022. arXiv: 2310.14821 [cs.DC].

[193] S. Das, S. Duan, S. Liu, A. Momose, L. Ren, and V. Shoup, *Asynchronous Consensus without Trusted Setup or Public-Key Cryptography*, 2024. IACR: 2024/677 (cs.DC).

[194] S. Ren, C. Lee, E. Kim, and S. Helal, "Flexico: An efficient dual-mode consensus protocol for blockchain networks", *PLoS ONE*, 2022.

[195] Y. Lu, Z. Lu, and Q. Tang, "Bolt-Dumbo Transformer: Asynchronous Consensus As Fast As the Pipelined BFT", in *CCS '22: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[196] T. S. team, *\NoCaseChange{Sui}*, https://github.com/mystenLabs/sui, 2024.

[197] E. Blum, J. Katz, J. Loss, K. Nayak, and S. Ochsenreither, "Abraxas: Throughput-Efficient Hybrid Asynchronous Consensus", in *CCS '23: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

Pasindu Tennage

Email: pasindu.tennage@gmail.com
LinkedIn: linkedin.com/in/pasindutennage
GitHub: github.com/PasinduTennage
Website: pasindutennage.github.io
Address: Lausanne, Switzerland

Skills and Expertise

- **Technical Skills**: Consensus, Blockchains, Distributed Algorithms, BFT, Cryptography, Scalability, Software Engineering, System Architecture, Software Testing, Performance Optimization.
- **Programming and Scripting Languages**: Rust, Go, Python, Java, C, SQL, Latex, Bash.
- **Tools**: AWS EC2, Git, Linux, NetEm.

Professional Experience

- Research Assistant 03/2020 – present
  - **Decentralized and Distributed Systems Laboratory, EPFL**, Lausanne, Switzerland
  - Focused on advancing scalable consensus protocols by developing robust and high performance consensus protocols and contributing to protocol design, implementation, and performance evaluation.
  - Collaborated with Mysten Labs on industry projects, contributing to the design and implementation of a novel secure distributed consensus protocol.
  - Conducted high-quality, impactful research, with publications in ACM SOSP, demonstrating strong verbal and written communication skills.
- Scientific Researcher 03/2022 – 08/2022
  - **Institute of Science and Technology (ISTA)**, Austria
  - Designed and implemented a novel consensus protocol and a scalable overlay network, both optimized for high-performance and scalability.
- Research Assistant 09/2019 – 03/2020
  - **Dependable Systems Lab, EPFL**, Lausanne Switzerland.
  - Designed, developed, and tested a custom concurrent hash table in C to reduce Network Address Translation latency, ensuring robustness, scalability, and high performance.
- Software Engineer 05/2018 – 08/2019
  - CTO office, **WSO2**, Sri Lanka
  - Led the development of a new workload characterization suite for Java microservices, enhancing the understanding of performance across various system parameters.
  - Conducted high-quality and impactful research, with publications in top-tier conferences such as HPCC and ICWE, showcasing strong verbal and written communication skills.
- Intern Software Engineer 07/2016 – 12/2016
  - Identity server team, **WSO2**, Sri Lanka
  - Led the implementation of a new UI feature for the WSO2 Identity Server that was later included in WSO2's commercial product.

Education

- Ph.D. Computer and Communication Sciences 09/2019 – 03/2025
  - **École Polytechnique Fédérale de Lausanne** (**EPFL**), Switzerland
- Master's Degree, Computer Science and Engineering (GPA: **4.2**/**4.2**) 02/2018 – 08/2019
  - **University of Moratuwa**, Colombo, Sri Lanka
- Bachelor's Degree, Computer Science and Engineering (GPA: **4.13**/**4.2**) 02/2014 – 12/2017
  - **University of Moratuwa**, Colombo, Sri Lanka
  - Awarded the Gold medal for the highest GPA among 700 engineering undergrads across all disciplines at the University of Moratuwa.
  - Awarded the Mahapola Scholarship.

Personal Information

- **Nationality**: Sri Lankan
- **Languages**: English (Fluent), French (Knowledgeable), Sinhalese (Native)