Optimising Order-Fairness in Blockchains using Ordering Linearizability

Pouriya Zarbafian

A thesis submitted to fulfil requirements for the degree of

Doctor of Philosophy

School of Computer Science

Faculty of Engineering

The University of Sydney

November 2024

Statement of Originality

This is to certify that to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any degree or other purposes.

I certify that the intellectual content of this thesis is the product of my own work and that all the assistance received in preparing this thesis and sources have been acknowledged.

Pouriya Zarbafian
November 1, 2024

Authorship Attribution

The results presented in this dissertation were published in the following publications and can be found in the relevant chapters:

(1) Chapter 3 of this dissertation contains material under submission. A brief announcement of the material is published as [1]. I was the primary author of both materials by proposing the ideas and performing the analysis and experimentation.

[1] Pouriya Zarbafian and Vincent Gramoli. "Brief Announcement: Ordered Reliable Broadcast and Fast Ordered Byzantine Consensus for Cryptocurrency". In: *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*. Ed. by Seth Gilbert. Vol. 209. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 63:1–63:4. DOI: 10.4230/LIPICS.DISC.2021.63. URL: https://doi.org/10.4230/LIPIcs.DISC.2021.63

(2) Chapter 4 of this dissertation is published as [2]. I was the primary author who proposed the concept, performed the analysis, and did the experimental evaluation.

[2] Pouriya Zarbafian and Vincent Gramoli. "Lyra: Fast and Scalable Resilience to Reordering Attacks in Blockchains". In: *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023*. IEEE, 2023, pp. 929–939. DOI: 10.1109/IPDPS54959.2023.00097. URL: https://doi.org/10.1109/IPDPS54959.2023.00097

(3) Chapter 5 of this dissertation is published as [3]. Zhenliang and I were the primary authors who proposed the concept and performed the analysis.

[3] *Note: Authors in this publication are listed in alphabetical order.* Vincent Gramoli, Zhenliang Lu, Qiang Tang, and Pouriya Zarbafian. "AOAB: Optimal and Fair Ordering of Financial Transactions". In: *54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2024, Brisbane, Australia, June 24-27, 2024*. IEEE, 2024, pp. 377–388. DOI: 10.1109/DSN58291.2024.00045. URL: https://doi.org/10.1109/DSN58291.2024.00045

(4) Chapter 6 of this dissertation is published as [4]. I was the primary author who proposed the concept and performed the analysis.

[4] Pouriya Zarbafian and Vincent Gramoli. "Aion: Secure Transaction Ordering Using TEEs". In: *Computer Security - ESORICS 2023 - 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25-29, 2023, Proceedings, Part IV*. ed. by Gene Tsudik, Mauro Conti, Kaitai Liang, and Georgios Smaragdakis. Vol. 14347. Lecture Notes in Computer Science. Springer, 2023, pp. 332–350. DOI: 10.1007/978-3-031-51482-1_17. URL: https://doi.org/10.1007/978-3-031-51482-1%5C_17

(5) Chapter 7 of this dissertation is accepted for publication as [5]. Zhenliang and I were the primary authors who proposed the concept and performed the analysis.

[5] *Note: Authors in this publication are listed in alphabetical order.* Vincent Gramoli, Zhenliang Lu, Qiang Tang, and Pouriya Zarbafian. "Resilience to Chain-Quality Attacks in Fair Separability". In: *Computer Security - ESORICS 2024 - 29th European Symposium on Research in Computer Security, Bydgoszcz, Poland, September 16-20, 2024, Proceedings, Part IV*. ed. by Joaquín García-Alfaro, Rafal Kozik, Michal Choras, and Sokratis K. Katsikas. Vol. 14985. Lecture Notes in Computer Science. Springer, 2024, pp. 251–270. DOI: 10.1007/978-3-031-70903-6_13. URL: https://doi.org/10.1007/978-3-031-70903-6%5C_13

In addition to the statements above, in cases where I am not the corresponding author of a published item, permission to include the published material has been granted by the corresponding author.

Pouriya Zarbafian
Signature: *Date:* November 1, 2024

As supervisor for the candidature upon which this dissertation is based, I can confirm that the authorship attribution statements above are correct.

Associate Prof. Vincent Charles Gramoli
Signature: *Date:* November 1, 2024

Acknowledgements

The lengthy road leading to the completion of this PhD has been filled with challenges, discoveries, and growth. It is with immense gratitude that I reflect on the support and encouragement of those who have made this journey possible.

First and foremost, I would like to extend my deepest thanks to my supervisor, Vincent. His incredible guidance, selfless availability, and constant support have surpassed all my expectations. This journey has been profoundly shaped by his knowledge and dedication.

I would like to express my gratitude to Professor Nicolas Guelfi. His unwavering support and encouragement at the outset of this journey were instrumental in helping me start my PhD.

I want to thank my long-time friend, Guillaume, for always being there when I needed it the most and being a guiding light through all these years.

I would like to thank Zhenliang and Pierre, two postdoc students with whom I had the honour of working. Their knowledge, methodology, and patience have greatly enhanced the quality of my work.

Finally, I would like to thank Chris for his immense generosity and continuous support. His kindness and dedication are a constant source of inspiration.

Abstract

The world of digital finance was forever reshaped by the release of Bitcoin, seen as a symphony of cryptographic and distributed systems principles orchestrated together to provide a coherent machine for permissionless, peer-to-peer payments. This singular event led to the emergence of a new paradigm where a ledger of digital payments could be upheld by completely decentralized participants of a protocol that could freely join and leave, and contrasted immensely against traditional centralized finance models. The success of this new paradigm brought great attention to its underlying technology, State Machine Replication (SMR), but exposed a shortcoming in the way transactions were ordered in SMR protocols. Although a slight detail, the shortcomings of ordering were misused by malicious actors to reorder transactions and exploit hundreds of millions of profit from unsuspecting users.

To mitigate the risk of these shortcomings, the idea of order-fairness was introduced to constrain the output of SMR so that the final ordering of transactions reflects the ordering observed by all participating processes. Ordering linearizability is one ordering technique that achieves such a result. Specifically, with ordering linearizability, processes assign ordering indicators to transactions they observe, ordering the output using the indicators that are upper-bounded and lower-bounded by ordering indicators assigned by correct processes. However, ordering linearizability is not sufficient to prevent all reordering attacks. Furthermore, its implementation induces additional costs compared to standard SMR and displays sub-optimal time and communication complexity. This dissertation builds upon ordering linearizability and introduces SMR protocols that improve resilience to reordering attacks, reduce the additional costs induced by adding order-fairness, and fill the gaps present in the formulation of ordering linearizability.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Decentralized Finance can be regarded as one of the largest technical crossovers with financial systems the world has seen in the last decade, provoking new ways of thinking and fostering some of the largest shifts of wealth observed since the .com boom. The singular event that instantiated this phenomenon was the introduction of Bitcoin [6], by simply allowing two peers to exchange various digital assets without being encumbered by a necessary trusted third-party, as is customary in traditional centralized finance. The founding concepts of Bitcoin build heavily upon the works of State Machine Replication (SMR), the technique of sharing state amongst a set of processes. Prior to Bitcoin's innovative techniques, most SMR required a predetermined set of participating processes, or at the minimum, knowledge of all participating processes. This allowed SMR to operate safely with a quantitative threshold that the strict majority of processes are correct and following the protocol [7]. In fact, an adversary controlling a majority of the processes can lead to two distinct correct processes reaching different decisions.

Switching to an SMR system where participating processes can freely join and leave poses significant technical challenges. These challenges are superimposed with the challenge of identity, where designing an open system requires achieving resilience to Sybil attacks [8], the scenario in which an attacker assumes the identities of a superficially large number of processes in order to dictate the decisions made by correct processes. Bitcoin ingenuously combined proofs of cryptographic work [9] to SMR, and thereby solved the problem of resilience to Sybil attacks in an open system where processes can dynamically join and leave. The feasibility of trustless digital transfers in open systems opened the prospect of fully decentralized applications and led to the wide success of blockchain technology and decentralized finance. Interestingly, this success also put forward shortcomings in the SMR specification.

In the context of digital asset transfer, the SMR specification requires that each process outputs the same ordered set of transactions. However, it does not specify which orderings are valid. Most blockchains are implemented using SMR and, as a result, are vulnerable to reordering attacks. In fact, the lack of ordering requirements in the SMR specification has enabled malicious participants to perform various reordering attacks. Daian et al. [10] were the first to investigate these new forms of attacks in Ethereum [11]. To address these attacks, Kelkar et al. [12] proposed to extend the SMR specification with order-fairness so that the final ordering of transactions reflect the ordering observed by processes. Following, Zhang et

al. [13] introduced a novel paradigm for order-fairness named *ordering linearizability*. Ordering linearizability circumvents the cyclic dependencies that may arise with other implementations of order-fairness. Hence, in this dissertation, we adopt the ordering linearizability paradigm for order-fairness.

Unfortunately, existing implementations of order-fairness present multiple shortcomings. First, the proposed solutions require additional message delays before committing a transaction, resulting in an increased cost in communication. On its own, an increased commit latency directly affects the finalization of a transaction. Second, these solutions display a cost per transaction that can potentially be cubic in the number of processes, and this diminishes their scalability and their resilience to adversarial scenarios. The additional costs of order-fairness, in turn, have prevented its large-scale adoption in SMR. Second, existing implementations of order-fairness are not resilient to all reordering attacks. For instance, because of the violation of triangle inequalities in network delays [14], order-fairness approaches based solely on the orderings observed by individual processes [12, 13] are still vulnerable to front-running attacks [15, 16]. Correspondingly, order-fairness approaches that simply obfuscate the payloads of transactions [17] are at the mercy of a Byzantine leader enforcing an arbitrary ordering or censoring some of the transactions.

This dissertation aims to optimise order-fairness by reducing any additional overhead incurred whilst improving its resilience to reordering attacks. First, by providing order-fairness at almost no extra cost, we aim at facilitating the integration of order-fairness into SMR protocols. Second, the resilience of order-fairness to reordering attacks is improved by combining (i) the orderings observed by correct processes with (ii) payload obfuscation and (iii) decentralized protocols for coordination. Finally, we analyze shortcomings in the definition of ordering linearizability and devise protocols to strengthen against the observed results.

1.1 Objectives

In this section, we first present the research objectives of this dissertation. To this end, we first articulate the perimeter of our research, and then divide this perimeter into three objectives. The main objective of this dissertation is to

optimise order-fairness in blockchains using ordering linearizability.

The optimisation of order-fairness can be embodied into three distinct aspects. First, in Section 1.1.1, we present our objective to reduce the cost incurred by adding ordering linearizability to SMR. Second, in Section 1.1.2, we present our objective to improve the resilience of ordering linearizability to reordering attacks. Third, in Section 1.1.3, we present our objective to strengthen the ordering linearizability paradigm.

1.1.1 Objective 1: Reduce the Cost of Ordering Linearizability

The existing implementation of ordering linearizability [13] requires at least 11 message delays to commit a transaction. Furthermore, all existing implementations of order-fairness [12, 13,

16, 18] have a communication complexity of $\mathcal{O}(n^3)$ whereas the communication complexity to commit a transaction without fair ordering is only $\mathcal{O}(n^2)$ [19], a linear multiplicative factor less. We thus enunciate the two following sub-objectives:

1.1 To reduce the commit latency of ordering linearizability, and in particular, to find the associated lower bound.

1.2 To reduce the communication complexity of ordering linearizability and find if ordering linearizability can be achieved with an $\mathcal{O}(n^2)$ communication complexity.

The goal of this objective is to research lower bounds associated with ordering linearizability and to find implementations of ordering linearizability that are optimal. By reducing the costs associated with adding ordering linearizability to SMR, we reduce the obstacles to the adoption of order-fairness in SMR.

1.1.2 Objective 2: Improve Resilience to Reordering Attacks

Implementations of order-fairness based on the ordering of transactions observed by processes are still vulnerable to reordering attacks such as front-running [16]. On the other hand, implementations purely based on payload obfuscation [17] do not prevent an arbitrary order or omission by a Byzantine leader. We separate the objective of resilience to reordering attacks into two sub-objectives:

2.1 To analyse shortcomings and vulnerabilities to reordering attacks in existing implementations of order-fairness.

2.2 To devise an implementation of order-fairness that improves on resilience to reordering attacks.

The initial goal of order-fairness was to prevent reordering attacks. However, with current solutions, this goal has only been partially achieved. With this objective, our goal is to help fill the gap by devising protocols that improve on resilience to reordering attacks.

1.1.3 Objective 3: Strengthen Ordering Linearizability

First, ordering linearizability relies on ordering indicators assigned to transactions by processes. Consequently, sending processes that are isolated due to the network disposition are disadvantaged. Second, ordering linearizability only ensures weakened validity and ordering safety properties. For instance, an asynchronous network can cause transactions submitted by correct processes to be discarded or to be ordered with an order that is different than the one perceived by correct processes. Even though a stronger definition of ordering linearizability denoted *fair separability* [16] has been proposed to address these issues, the only existing implementation of fair separability [20] is vulnerable to attacks on chain-quality whereby Byzantine processes monopolise the set of committed transactions. This vulnerability is due to the fact that in [20], a transaction must be output even if it has been observed by a *single* correct process. Note that

ordering linearizability only applies if a transaction has been observed by *all* correct processes. Thus, we divide this research objective into two separate sub-objectives:

3.1 To analyse the shortcomings resulting from the definition of ordering linearizability.

3.3 To devise protocols that address these shortcomings and fix the network bias and the weakened validity and ordering safety properties of ordering linearizability.

In its initial formulation, ordering linearizability lacks completeness as it weakens validity in atomic broadcast and its ordering safety property can sometimes vanish. Our third objective focuses on strengthening ordering linearizability by reducing its network bias and by strengthening its validity and ordering safety properties. This, in turn, strengthens ordering linearizability by rendering it more fair.

1.2 Contributions

In this dissertation, we optimise order-fairness in blockchains by presenting five novel contributions. First, we introduce broadcast protocols that achieve ordering linearizability with minimal latency and optimal communication complexity. Second, we devise SMR protocols that improve resilience to reordering attacks. Finally, we present protocols with strengthened formulations of ordering linearizability.

Specifically, we present in this dissertation the five following contributions:

1. **Fast Ordered Byzantine Consensus.** The implementation of ordering linearizability presented in [13] requires 11 message delays to commit a transaction because it sequentially executes an ordering phase followed by a consensus phase. However, without ordering linearizability, a transaction can, in theory, be committed after only 3 message delays [21]. To decrease the number of message delays, we introduce new protocols that piggyback ordering data and parallelise the ordering and the consensus step. Reducing latency reduces the costs incurred by ordering linearizability and thus follows **Objective 1**.

2. **Lyra: Fast and Scalable Resilience to Reordering Attacks.** Achieving ordering linearizability without adding message delays is impossible if the final ordering indicator results from a set of collected ordering indicators. This impossibility results from the fact that after the set has been collected, at least 3 additional message delays are required to decide an ordering indicator for the transaction [21]. Therefore, we devise a new approach where processes exchange information about network latencies to predict ordering indicators and present a protocol that provably has minimal latency. In addition, we also implement payload obfuscation in a leaderless protocol so that our protocol is resilient to reordering attacks. Achieving ordering linearizability with minimal latency follows **Objective 1**, whereas achieving resilience to reordering attacks follows **Objective 2**.

3. **AOAB: Optimal and Fair Ordering of Financial Transactions.** In existing implementations of order-fairness, the communication complexity for each transaction is $\mathcal{O}(n^3)$, which is greater than the $\mathcal{O}(n^2)$ communication complexity for outputting a transaction in classical SMR. In addition, when processes assign timestamps to a transaction, and the sender of the transaction collects this set of timestamps, the set of collected timestamps may expire due to asynchrony. This requires the sender to collect another set of timestamps. With this contribution, we introduce a protocol that achieves ordering linearizability with an optimal communication complexity of $\mathcal{O}(n^2)$ bits per transaction, and this improvement advances **Objective 1**. The fact that our protocol uses a commit-reveal scheme to hide the payloads of transactions improves resilience to reordering attacks and supports **Objective 2**. Finally, by ensuring that the transactions submitted by correct processes never expire and are eventually committed follows **Objective 3**.

4. **Aion: Secure Transaction Ordering using TEEs.** Ordering linearizability relies on ordering indicators assigned by processes to the transactions that they observe. The receive time of a transaction is thus intrinsically dependent on the distances between processes and creates a bias for sender processes that are geographically isolated. To address this bias, we leverage trusted execution environments and combine secure time with the knowledge of network distances to enable processes to determine the time when a transaction was sent. This, in turn, removes the bias introduced by the network disposition. By devising a protocol that sequences transactions at the time when they are sent, we improve resilience to reordering attacks, which supports **Objective 2**. Strengthening the ordering paradigm of ordering linearizability meets **Objective 3**.

5. **Resilience to Chain-Quality Attacks in Fair Separability.** The ordering linearizability paradigm only applies to committed transactions. Hence, the ordering constraint simply vanishes if two transactions observed by processes in a specific order are not committed. To address this shortcoming, we devise a protocol that ensures the unconditional application of the ordering linearizability paradigm to all the transactions observed by correct processes. This improvement directly strengthens ordering linearizability and thus meets **Objective 3**.

1.3 Outline

The rest of this dissertation is organised as follows:

- Chapter 2 first introduces related work and concepts relevant to this dissertation and then presents our computational model.

- Chapter 3 proposes our leaderless implementation of ordering linearizability with reduced latency. We also present an experimental evaluation of our protocol and show that despite using a leaderless protocol, a direct implementation or ordering linearizability does not scale.

- Chapter 4 presents Lyra, an implementation of ordering linearizability with optimal latency. To improve latency and scalability, we introduce a novel approach that replaces signed timestamps with the knowledge of network delays. We evaluate Lyra on a geo-distributed setup and show that it scales up to a hundred nodes while significantly improving the performance of existing solutions.
- Chapter 5 proposes AOAB, an implementation of ordering linearizability that provides a stronger liveness guarantee and has optimal communication complexity. Contrary to previous implementations, this protocol ensures that transactions broadcast by correct processes are never discarded.
- Chapter 6 introduces Aion, a novel approach to fair ordering that eliminates the network bias in ordering linearizability. Aion leverages trusted execution environments to order transactions at the time that they are broadcast, thus achieving instant ordering.
- Chapter 7 presents a protocol that implements fair separability, a stronger notion than ordering linearizability. Fair separability relies on the same ordering paradigm as ordering linearizability but applies to all transactions.
- Chapter 8 concludes this dissertation and suggests directions for future work.

Chapter 2

Background

This chapter presents concepts and works related to the ideas explored in this dissertation. We begin by presenting related works surrounding the topics of broadcast and consensus protocols and outline how these protocol abstractions form the basis of State Machine Replication (SMR). Following, we discuss concepts relating to blockchains based on SMR, reviewing attacks enabled by shortcomings in the SMR specification. We then discuss various approaches adopted to address these shortcomings, but we also call attention to potential hazards in their implementation. Finally, we present the model of computation used throughout the dissertation.

2.1 Related Work

2.1.1 Broadcast Protocols

The inherent nature of distributed systems implies that distributed protocols must be resilient to various types of failures. The concept of Byzantine failure [22] was introduced as an umbrella term to designate the fact that a Byzantine process, i.e., a process under the control of an adversarial entity, can deviate arbitrarily from the prescribed protocol. For instance, a Byzantine process may lie about its internal state or about the messages that it has received; it can equivocate by being duplicitous in the messages it sends, or it can simply pretend that it has crashed. As a result, broadcast abstractions used in distributed protocols must be designed to be resilient to the behaviours of Byzantine processes. One fundamental broadcast protocol is the Reliable Broadcast protocol [23], also denoted by Secure Broadcast [24]. Reliable broadcast enables the atomic delivery of a value, v, by all correct processes; that is, either all correct processes eventually deliver v, or none of them do. Multiple variants of the Reliable Broadcast protocol have been introduced.

Secure Causal Atomic broadcast [25] adds payload obfuscation to reliable broadcast using cryptography while, in parallel, ensuring that data payloads can later be rebuilt. Our Lyra protocol presented in Chapter 4 uses verifiable secret sharing [26] to achieve broadcast causality. Causal Broadcast [27], also denoted by First-In First-Out broadcast [28], is a variant of reliable broadcast where the messages sent by a process are delivered by correct processes with respect to their causal order [29]. In Chapter 7, we combine reliable broadcast with threshold signatures

to achieve properties similar to causal broadcast. We also introduce in Section 3.2 a variant of the reliable broadcast that delivers a set of timestamps with the delivered value. On the other hand, Total Order Broadcast [30] implements a total order of messages where all messages from all senders are delivered in the same order by each correct process.

The Binary Value Broadcast protocol was introduced in [31] and is a variant of reliable broadcast that applies to cases when the value broadcast is a binary value. Binary value broadcast is used as a building block for several binary consensus algorithms [31, 32, 33]. In [34], this protocol is enhanced with accountability in order to detect processes that do not follow the protocol. In this dissertation, we present two variants of the Binary Value Broadcast protocol. Our Order Value Broadcast (see Section 3.3.4) piggybacks ordering metadata, whereas our Validating Value Broadcast (see Section 4.3.1) adds a configurable validation function.

2.1.2 Agreement Protocols

Agreement, or consensus, protocols enable a set of processes to agree on a single value. In the presence of malicious, or Byzantine, processes [35, 22], agreement cannot be guaranteed unless the number of Byzantine processes is less than a third of the total number of processes [7]. If the network is completely asynchronous, meaning no upper bound on message delays, no agreement protocol can guarantee deterministic termination [36]. Randomized protocols [37, 38, 39] were introduced to ensure termination in asynchronous networks in the presence of Byzantine processes with overwhelming probability. In a partially synchronous network [40], where there exists a message delay that is unknown a priori, or there is a time by which the network becomes stable, termination of the protocol can be guaranteed in the presence of Byzantine processes. In this dissertation, except for Chapter 5, we only consider partially synchronous consensus protocols. Byzantine processes can also influence the cost of consensus. For instance, by equivocating or sending duplicitous messages, they can delay or thwart agreement between processes. In adversarial cases where there are n processes out of which $f < \frac{n}{3}$ are Byzantine, the lower bounds for time complexity, message complexity, and communication complexity are $f + 1$ [41], $\mathcal{O}(n^2)$ [19], and $\mathcal{O}(n^2)$, respectively.

Leader-based consensus protocols [40] rely on a designated leader to drive and coordinate the protocol. A leader-based consensus protocol with expected constant time is presented in [42]. If a leader misbehaves by crashing or exhibiting Byzantine behaviour, the throughput and latency of these protocols can be afflicted as a new leader must be designated. Furthermore, the leader may also impede the scalability of the operation, as it becomes a bottleneck having to handle inputs from all processes [43].

Leaderless consensus protocols, on the other hand, can solve consensus without requiring a designated leader. It should be noted, however, that some multi-leader algorithms [44, 45] cannot tolerate Byzantine failures. Borran and Schiper [46] detail a leaderless algorithm that has exponential complexity. Lamport suggests replacing the real leader of consensus algorithms by repeatedly invoking an election of a virtual leader but does not specify the algorithm [47, 48]. A randomized leaderless algorithm [31] has been used in blockchains [32, 49, 50]; however, an adversarial scheduler prevents it from converging towards a decision [51] and, despite some

patches, we are not aware of a simple fix that was proved correct. The leaderless DBFT protocol [33] was recently proved safe and live with model checking [52].

2.1.3 State Machine Replication

A State Machine Replication (SMR) protocol enables processes to agree on a totally ordered set of transactions [29]. Although SMR requires agreement, implementing SMR is more expensive than agreement [53] due to potential dependencies between the transactions that must be committed. The literature on SMR is extensive and addresses multiple aspects of this paradigm. A Byzantine fault-tolerant (BFT) SMR protocol is introduced in [54]. PBFT [55] presents a practical SMR protocol for the partially synchronous setting with optimal Byzantine resilience. Research in the field of BFT SMR entails optimisations to reduce the cost of SMR based on trade-offs and assumptions. Zyzzyva [56] reduces the cost by assuming first that the leader is correct. Aardvark [57] focuses on improving resilience in adversarial scenarios, whereas XFT [58] simplifies the design by focusing only on certain types of failures. Raft [59] aims at providing a simpler protocol to reduce implementation errors. SBFT [60] improves scalability by relying on fast paths and threshold signatures. HotStuff [61] increases throughput by pipelining successive consensus steps. In Section 4.4, we introduce a novel SMR protocol to satisfy additional requirements needed for a fair ordering of transactions.

2.1.4 Blockchain

To guarantee safety, BFT SMR protocols require an upper bound f on the number of Byzantine processes [7] and thus rely on a predetermined set of n processes that take part in the protocol, with $f < \frac{n}{3}$. Such setting is commonly denoted as *permissioned* as the identity of participating processes is known beforehand by all processes. By bringing proof-of-work [9] into SMR, Bitcoin [6] introduced an SMR protocol for the *permissionless* setting where processes can dynamically join and leave the protocol. In Bitcoin, the distributed ledger consists of a chain of ordered blocks where each block comprises a set of ordered transactions, thus implementing a total ordering of transactions. To be *mined*, i.e., be considered valid, each block requires work under the form of hashing. In essence, the idea was to give one vote per CPU and keep the longest chain of blocks. Hence, Bitcoin circumvents Sybil attacks [8] and preserves safety as long a majority of the hashing power is detained by honest processes.

The seminal work of Bitcoin [6] has led to the success of blockchain technology and to the widespread adoption of decentralized finance (DeFi) [62]. The security of this new consensus protocol, denoted by the Nakamoto consensus, has only been analysed a posteriori [63, 64]. The sleepy model of consensus [65] formalises a system where, at any time, some of the processes may be online, whereas others are offline. The centralization trends in voting power due to mining pools are analysed in [66]. To circumvent the lack of sustainability of proof-of-work based blockchains, proof-of-stake blockchains have been introduced [67, 68, 69] to replace a majority of hashing power by a majority of staked assets.

Blockchains leverage SMR to implement distributed ledgers that transfer digital assets and execute smart contracts, thus leading to the denomination of Distributed Ledger Technology

(DLT). The transfer of assets in DLT relies on appending to the ledger transactions from authenticated users [70]. On the one hand, the owner of the assets signs the transaction that it issues, and on the other hand, all processes verify the authenticity of the transaction by checking the ownership of the transferred assets in the ledger.

Permissioned blockchains [71, 72, 73, 74] consist of blockchains where the set of validator processes taking part in the consensus is restricted to a determined set. Although the set of validators can evolve, evolution is controlled by the set of existing validators. However, compared to classical SMR, the set of validators in permissioned blockchains regroup a set of heterogeneous processes resulting from a consortium of enterprises. Hence, permissioned blockchains are also called consortium blockchains. Furthermore, permissioned blockchains are open to transactions submitted by external clients. In this dissertation, we only consider the context of permissioned blockchains where the set of processes is known.

2.1.5 Cryptographic Challenges

Cryptographic challenges leverage the statistical properties of the outputs of cryptographic schemes to thwart Byzantine behaviours. Cryptographic challenges were introduced by Dwork and Naor [75] as a way to limit junk emails. They are commonly used as a way to prevent denial-of-service attacks in networks [76, 77, 78, 79]. One of the fundamental applications of cryptographic challenges in blockchains is to preserve safety and to mitigate the risk of Sybil attacks [8] whereby an attacker impersonates a large number of participants. Proofs of work were introduced by Hashcash [9] and are used in Bitcoin [6] to mine new blocks and extend the distributed ledger. Ouroboros [67] and Algorand [68] are based on proof-of-stake mechanisms that use verifiable random functions [80]. In our SMR protocol in Chapter 6, we leverage cryptographic challenges to ensure the freshness of the timestamps sent by processes.

2.1.6 Reordering and MEV Attacks

The SMR specification requires that each process order transactions in the same order. However, any order is considered valid, provided that it is accepted by all correct processes. This shortcoming in the SMR specification is identified in [81] and addressed in the SMR literature using payload-encryption [81, 25, 82]. Because most blockchains rely on SMR, they are also vulnerable to a wide range of transaction reordering attacks, and this lack of ordering requirement has been exploited by malicious users for profit [10]. The problem is revisited in the blockchain literature and the suggested solution is based on implementing a fair-ordering of transactions based on the orderings observed by processes [12]. Examples of reordering attacks include *front-running attacks*, which are systematised in [15]. A front-running attack consists of a transaction t' being ordered before a transaction t that was issued before t'. Such an attack enables, for instance, the issuer of t' to profit from a price increase induced by t. If t was buying a non-fungible token or a unique asset, then transaction t would simply become obsolete. Front-running first appeared in centralized finance [83] and is considered illegal by the Security and Exchange Commission (SEC) [84]. However, in blockchains, due to the fact that submitted transactions are public, malicious processes can, in fact, execute front-running

attacks. *Sandwich attacks* are another type of reordering attack. A sandwich attack consists of a transaction t being front-run by a transaction t_{before} and back-run by a transaction t_{after}. This type of attack usually occurs in decentralized exchanges (DEX) whereby t_{before} buys from the same asset as t, thus profiting from the price increase that will be induced by t, and t_{after} sells the assets bought by t_{before}, cashing in on the profit generated by t. Finally, by omitting or censoring a transaction t, t is reordered. Even though the issuer of t may resubmit its transaction, t ends up being ordered at a later time.

By extension, Miner Extractable Value (MEV) consists of all the attacks where transactions are either reordered or inserted in or excluded from the output. A taxonomy of the existing MEV attacks and countermeasures is presented in [85]. The first investigation of MEV attacks is presented by Daian et al. [10]. It has been shown that these attacks have ripped honest users from hundreds of millions of dollars of profit [86]. In [87], it is estimated that between 2021 and 2022 alone, more than USD 675M of MEV has been extracted. More recent work on MEV relies on the analysis of transactions and smart contracts in order to find vulnerabilities. In [88], Zhou et al. devise methods to automatically create profit-generating transactions and study how such behaviours can incentivise forks in blockchains. By formalizing smart contracts, [89] presents a tool for the automatic analysis of values that can be extracted from smart contracts. In [90], Heimbach et al. present a game-theoretic model [91] based on incentives to prevent MEV attacks.

2.1.7 Order-Fairness

Time Order-Fairness

To prevent reordering and MEV attacks, the first proposed solution aims at implementing *time order-fairness* whereby transactions are ordered based on the time when they are received by processes. Input causality using timestamps assigned to transactions is introduced in [29] and extended to vector clocks for the asynchronous setup in [92] and [93]. These seminal works are adapted to the BFT SMR context under the denomination of order-fairness. There exist two paradigms for time order-fairness. The first paradigm is introduced by Kelkar et al. [12] and consists of ordering a transaction t_1 before a transaction t_2 if a majority of processes order t_1 before t_2. This ordering paradigm relies on building dependency graphs between transactions. We denote this first paradigm by *relative ordering* as the final ordering of a transaction is expressed as a relative ordering compared to other transactions. In [18], Cachin et al. show the lower bound for ordering two transactions t_1 and t_2 using relative ordering. Specifically, they build upon results in differential consensus [94] to compute the differential number in the preferences of processes to order, say, t_1 before t_2. Mu et al. [95] improve the performance of relative ordering by decoupling the ordering and the consensus phases.

The second paradigm for time order-fairness is introduced by Zhang et al. [13] and aims at circumventing the cyclic dependencies [96], also denoted by Condorcet cycles, that may arise when ordering transactions using relative ordering. This second paradigm assigns a unique ordering indicator to each transaction that is used to order the transaction. We denote the second paradigm by *absolute ordering* as the assigned ordering indicator is expressed independently of

the orderings of other transactions. In Basil [97], absolute ordering is implemented with respect to isolation levels. In this dissertation, we optimise absolute ordering to make it more practical and resilient. The implementation of absolute ordering in [13] does not apply to transactions that are not committed. A stronger definition of absolute ordering is introduced in [98] and called *fair separability* in [16]. Fair separability applies to all the transactions observed by correct processes. In [20], we introduce the first implementation of fair separability and make it optimal in communication complexity. In Chapter 7, we introduce an implementation of fair separability that is resilient to attacks on chain quality by showing that fair separability can be achieved without having to commit all the transactions from Byzantine processes.

Blind Order-Fairness

Despite extending the SMR specification with fair ordering, time order-fairness is insufficient to prevent reordering and MEV attacks [16]. Due to the violation of triangle inequalities in network delays [14], attackers can leverage a faster route and front-run the transactions that they observe (see Chapter 4). To prevent MEV, the payloads of transactions should only be revealed after their final orderings are determined [99]. This second approach is often called *blind order-fairness* and relies on an encryption-based scheme. Blind order-fairness is introduced in [81] where it is denoted by *causal order*. By obfuscating the payloads of transactions, causal order ensures that the input of other transactions cannot depend on the transactions that have been submitted but not yet committed. Causal order is implemented for SMR in HoneyBadger [32], and in Dumbo [100] and sDumbo [101] with optimal communication complexity. In [102], Asayag et al. combine a commit-reveal scheme with a random ordering of transactions. To prevent MEV in blockchains, Malkhi and Szalachowski [17] implement blind order-fairness using a commit-reveal scheme based on secret-sharing [103]. The protocols presented in Chapter 4 and Chapter 5 combine blind order-fairness with time order-fairness to improve resilience to MEV attacks.

2.1.8 Trusted Execution Environment

A Trusted Execution Environment (TEE) is a type of hardware that provides secure and restricted access to its stored data. It also provides the ability to remotely verify the code that is executed by the TEE. Stathakopoulou et al. [104] use TEEs to add fairness to the ordering of transactions. To prevent front-running attacks, their approach consists of obfuscating transactions until they are committed and delegating the actual ordering to the total broadcast layer. Therefore, their approach is closer to a commit-reveal scheme [99]. Gupta et al. [105] also leverage trusted components in SMR but focus instead on improving liveness and reducing communication complexity, and although their protocol supports concurrent executions of consensus, it does not implement order-fairness. In blockchains, multiple protocols rely on TEEs to implement SMR. TEEs are used to improve scalability [106], limit the behaviour of Byzantine processes [107], or secure off-chain transactions [108, 109], but they do not consider fairness in the ordering of transactions. The SMR protocol presented in Chapter 6 leverages TEE to assign secure timestamps to transactions by computing safe values of network delays.

2.2 Computational Model

2.2.1 Processes

In this dissertation, we consider a system of n processes denoted by $P = \{p_1, \ldots, p_n\}_{i \in [n]}$. We assume that each process is sequential and thus executes each step of the protocol after the other. We denote by *correct* processes that follow the prescribed protocol and denote by *Byzantine* or *corrupt* processes that deviate from the protocol. We consider the classical Byzantine behaviour model where corrupt processes can deviate from the protocol arbitrarily [22]. We denote by P^{Corr} the set of correct processes and by P^{Byz} the set of Byzantine processes.

2.2.2 Adversary

We consider the faulty processes to be under the full control of a static adversary [25, 32]. This adversary model implies that prior to the start of the protocol, the adversary can select f processes to be completely corrupted. The adversary has access to the internal states of the corrupt processes and can manipulate their behaviours arbitrarily throughout the execution of the protocol. However, we assume that the adversary is polynomially bounded and that it cannot break the security of cryptographic schemes.

2.2.3 Network

We assume a *partially synchronous* network [40]. In the partially synchronous model, the network behaves asynchronously up to a *global stabilization time* (GST). The value of GST is unknown. After GST, the network behaves synchronously, and all messages between correct processes are delivered within a known bound Δ. We assume that the communication channels are reliable and that every message sent by a correct process is eventually delivered. In addition, we assume that channels are authenticated and that Byzantine processes cannot impersonate correct processes or tamper with the messages sent by correct processes. Finally, each process divides its execution into asynchronous rounds as defined in Dwork et al. [40].

2.2.4 Cryptography

Digital Signatures

A digital signature (DS) scheme [110] enables processes to sign their messages and allows other processes to verify the authenticity of signed messages. A DS scheme consists of the following algorithms.

- $\{sk_i\}_{i \in [n]}, \{pk_i\}_{i \in [n]} \leftarrow \mathsf{DS.Setup}(n, 1^\lambda)$. Given the number of processes n and a security parameter λ as inputs, this algorithm produces the set of public keys $\{pk_i\}_{i \in [n]}$ and the set of secret keys $\{sk_i\}_{i \in [n]}$ for the processes.

- $\sigma \leftarrow \mathsf{DS.Sign}(sk_i, v)$. This algorithm takes the secret key sk_i of a process p_i and a value v as inputs and outputs a signature σ for the value v.

- $0/1 \leftarrow \mathsf{DS.Verify}(pk_i, \sigma, v)$. This algorithm takes as inputs the public key pk_i of a process p_i, a signature σ, and a value v, and it outputs whether the signature was generated by p_i's secret key for the value v.

Threshold Signatures

An (f, n) threshold signature (TS) scheme [111] enables a set of n processes to create signature shares of a message. Then, $f + 1$ signature shares can be combined into a full signature for the message. A TS scheme consists of the following algorithms.

- $\{tsk_i\}_{i\in[n]}, \{tpk_i\}_{i\in[n]}, vk \leftarrow \mathsf{TS.Setup}(n, f, 1^\lambda)$. Given the number of processes n, the threshold f, and a security parameter λ as inputs, this algorithm produces a set of threshold public keys $\{tpk_i\}_{i\in[n]}$, a set of threshold secret keys $\{tsk_i\}_{i\in[n]}$, and a public verification key vk.

- $\pi \leftarrow \mathsf{TS.SignShare}(tsk_i, v)$. This algorithm takes the threshold secret key tsk_i of a process p_i and a value v as inputs and outputs a signature share π for the value v.

- $0/1 \leftarrow \mathsf{TS.VerifyShare}(tpk_i, \pi, v)$. This algorithm takes the threshold public key tpk_i of a process p_i, a signature share π, and a value v as inputs, and it outputs whether the signature share was generated for v using p_i's threshold secret key tsk_i.

- $\Pi \leftarrow \mathsf{TS.Combine}(\{tpk_i\}, \{\pi_i\}, v)$. This algorithm takes the threshold public keys of processes, a set of valid signature shares for v of size $f + 1$, and the value v as inputs and outputs a full signature for the value v.

- $0/1 \leftarrow \mathsf{TS.Verify}(vk, \Pi, v)$. This algorithm takes the verification key vk, a full signature Π, and a value v as inputs and outputs whether the full signature is valid for the value v.

Threshold Encryption

A (f, n) threshold encryption (TE) scheme [112] enables a set of n processes to encrypt a message and then generate decryption shares for the encrypted message. The encrypted message requires $f + 1$ decryption shares to be decrypted. A TE scheme consists of the following algorithms.

- $\{tsk_i\}_{i\in[n]}, \{tvk_i\}_{i\in[n]}, pk \leftarrow \mathsf{TE.Setup}(n, f, 1^\lambda)$. On input the number of processes n, the threshold f, and the security parameter λ, this algorithm generates the set of private-key shares $\{tsk_i\}_{i\in[n]}$, the set of verification key-shares $\{tvk_i\}_{i\in[n]}$, and the encryption key pk.

- $c \leftarrow \mathsf{TE.Encrypt}(pk, v)$. This algorithm takes as inputs the encryption key pk and a value v and outputs a cipher c consisting of the encrypted value of v.

- $\rho \leftarrow \mathsf{TE.DecryptShare}(pk, tsk_i, c)$. This algorithm takes as inputs the public key pk, the private-key share tsk_i of p_i, and a cipher c, and outputs a decryption share ρ for the cipher c.

- $0/1 \leftarrow \mathsf{TE.VerifyShare}(pk, tvk_i, c, \rho)$. This algorithm takes as inputs the public key pk, the verification key-share tvk_i of p_i, a cipher c, and a decryption share ρ, and outputs 1 if ρ is a valid decryption share of the ciphertext c by tsk_i and 0 otherwise.
- $v \leftarrow \mathsf{TE.Decrypt}(pk, c, \{\rho\})$. This algorithm takes as inputs the public key pk, a cipher c, and a set of decryption shares $\{\rho\}$ for c, and outputs a value v consisting of the decryption of the cipher c.

2.2.5 State Machine Replication

For each process $p_i \in P$, let Log_i denote the local output of the SMR protocol at p_i, i.e., the set of committed transactions.

Definition 2.1. *A protocol implementing an SMR ensures the following properties.*

- ***Safety.*** *For any two correct processes p_i and p_j, either Log_i is a prefix of Log_j, or Log_j is a prefix of Log_i.*

$$\forall p_i, p_j \in P^{Corr}, k = \min(|\mathsf{Log}_i|, |\mathsf{Log}_j|), \mathsf{Log}_i[1..k] = \mathsf{Log}_j[1..k]$$

- ***Liveness.*** *Eventually, the protocol outputs transactions.*

It follows from this definition that if an algorithm implements an SMR protocol, then when a transaction t is added to the Log_i of process p_i, it is also eventually added to the Logs of all correct processes. Thus, in this dissertation, we denote by $t \in \mathsf{Log}$ to say that t is committed.

$$t \in \mathsf{Log} \Leftrightarrow \Box(\forall p_i \in P^{Corr}, t \in \mathsf{Log}_i)$$

In contrast with relative ordering algorithms that rely on the ordered sequences of transactions observed by processes to deduce dependencies between transactions, absolute ordering algorithms assign a unique ordering indicator to each transaction that is output. Transactions that are output are then ordered using their respective ordering indicators. In this dissertation, all the ordering indicators used are in $\mathbb{N}$.

Definition 2.2 (Partial Order)**.** *A transaction t_1 output with an ordering indicator s_1 is executed before a transaction t_2 with an ordering indicator s_2 if $s_1 < s_2$. We say that t_1 is ordered before t_2 and write $t_1 \prec t_2$.*

2.2.6 Ordering Linearizability

Ordering Indicators

The goal of ordering linearizability is to order transactions based on the order perceived by correct processes. To this end, whenever a correct process p_i *receives* a transaction t, it assigns an *ordering indicator* s to t using the function Rank. In this dissertation, the assigned ordering indicator s can be either a timestamp or a sequence number. Let $\mathcal{T}$ denote the set of all possible transactions. The *local* function Rank_i assigning ordering indicators for process p_i is defined as:

$$\mathsf{Rank}_i \colon \mathcal{T} \longrightarrow \mathbb{N},$$
$$t \longmapsto s = \mathsf{Rank}_i(t).$$

Let also MaxRank (resp. MinRank) denote the function returning the highest (resp. lowest) ordering indicator assigned to a transaction t by any *correct* process.

$$\mathsf{MaxRank}(t) = \max_{\forall p_i \in P^{Corr}} (\mathsf{Rank}_i(t)),$$
$$\mathsf{MinRank}(t) = \min_{\forall p_i \in P^{Corr}} (\mathsf{Rank}_i(t)).$$

In this dissertation, we rely on two different implementations of the Rank function.

- Rank = SeqNum: each process uses a sequence number that it increments after assigning an ordering indicator to a transaction. The Rank function returns strictly monotonically increasing sequence numbers in $\mathbb{N}$.
- Rank = Clock: each process uses a real-time clock that returns strictly monotonically increasing timestamps in $\mathbb{N}$. The timestamp returned corresponds to the value read on the local clock.

Chapter 3, Chapter 4, and Chapter 6 use Rank = Clock, whereas Chapter 5 and Chapter 7 use Rank = SeqNum.

Ordering Linearizability

First introduced in [13], *ordering linearizability* is a correctness condition that extends the SMR specification by adding constraints to the ordering of the transactions output by an SMR. Ordering linearizability requires that if the highest ordering indicator assigned by a correct process to a transaction t_1 is lower than the lowest ordering indicator assigned by a correct process to a transaction t_2, and that both t_1 and t_2 are committed, then t_1 is executed before t_2.

Definition 2.3 (Ordering Linearizability)**.** *The output of an SMR satisfies* ordering linearizability *if and only if*

$$\forall (t_1, t_2) \in \mathcal{T}, \mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2) \wedge (t_1, t_2) \in \mathsf{Log} \Rightarrow t_1 \prec t_2.$$

Thus, ordering linearizability is a new safety property that applies to the output of an SMR. Ordering linearizability can be implemented by assigning to a transaction t an ordering indicator that is the median value of a set of $2f+1$ ordering indicators assigned to t by distinct processes. This ensures that the median value of this set is *correctly bounded*, i.e., it is upper bounded and lower bounded by the values of ordering indicators assigned to t by correct processes.

Definition 2.4 (Correctly Bounded)**.** *An ordering indicator s assigned to a transaction t is* correctly bounded *if and only if it is upper bounded and lower bounded by the ordering indicators assigned to t by correct processes.*

$$s \textit{ correctly bounded} \Leftrightarrow \mathsf{MinRank}(t) \leq s \leq \mathsf{MaxRank}(t).$$

2.2.7 Notations

For an XY-Broadcast protocol, we say that a process p_i XY-Broadcasts a value v when p_i submits v to the XY-Broadcast protocol, and say that p_i XY-Delivers w when p_i outputs a value w from the XY-Protocol. We denote by $\mathsf{Send}(\text{TAG}, x)$, the operation of sending to a process a message that consists of a tag TAG and a payload x. We denote by $\mathsf{Broadcast}(\text{TAG}, x)$ the operation of sending the message (TAG, x) to all processes.

- $\Pi[id]$ represents an instance of the protocol Π with a session identifier id. $\Pi[id](x)$ denotes invoking the instance $\Pi[id]$ with input x.
- Array slice: $A[x..y] = \{A[i] \mid x \leq i \leq y\}$, in ascending order.
- Tuple: we denote by (x, y, z) the 3-tuple consisting of the ordered elements x, y, and z.
- ℓ denotes the bit length of a transaction.
- λ denotes the cryptographic security parameter, which encompasses the size of a (threshold) signature and the length of the hash value.
- $\mathsf{Median}(A)$: $(f+1)^{th}$ value of the set A sorted in ascending order.

Regarding the notation Median, ordering linearizability uses the median value of a set S of $2f+1$ ordering indicators, i.e., the actual median value of S. This ensures that the median value is correctly bounded. However, to achieve a correctly bounded value in Chapter 7, we use the $(f+1)^{th}$ value of a set of $f+1$ values. We thus use the Median function to keep a consistent notation across chapters.

Chapter 3

Fast Ordered Byzantine Consensus

The order of transactions in cryptocurrency can have significant financial implications. A user who can reorder transactions can, for instance, exploit the information it has about an upcoming transaction to make a profit. In decentralized exchanges, reordering attacks are exploited by arbitrage bots [10] and cost users hundreds of millions of dollars [86]. Zhang et al. [13] proposed *ordering linearizability* in the best paper of OSDI '20 to cope with this problem. They propose an SMR protocol, called Pompē, that orders transactions before agreeing on them. One could possibly build a blockchain upon this protocol by using financial transactions and adding some validation phase to verify that each transaction is correctly signed and sufficiently funded. Unfortunately, Pompē requires at least 11 message delays to commit each transaction. Thus, this new blockchain would require at least twice as many message delays than recent blockchains like the Red Belly Blockchain that can commit transactions in only 4 message delays [74]. What remains unclear is whether one can devise a blockchain protocol that ensures ordering linearizability but requires fewer message delays to commit transactions.

The existing solution to ordering linearizability [13] is to first assign to a transaction a correctly bounded ordering indicator, one that falls in the interval of timestamps proposed by correct processes, and then to run a leader-based consensus protocol to decide upon this transaction and its associated ordering indicator. As the algorithm builds upon HotStuff [61] and SBFT [60], the resulting algorithm adds up to their time complexity. Hence, ordering linearizability is offered in 11 message delays after global stabilization time (GST). In this chapter, we present a fast algorithm for achieving ordering linearizability. To improve latency, we introduce a variant of the Reliable Broadcast protocol [113] that piggybacks timestamps so that transactions are delivered with a set of $2f + 1$ signed timestamps. We then derive the DBFT consensus algorithm from [33] and its underlying BV-Broadcast protocol [31] to validate the timestamps associated with transactions. We also provide an experimental evaluation of the proposed solution.

In summary, in this chapter, we present the following work:

1. We propose an Ordered Reliable Broadcast protocol where the broadcaster associates a timestamp to its transactions, which it broadcasts along with the transaction. Even

during an all-to-all execution of this ordered reliable broadcast, ordering linearizability is ensured—the transactions will end up being ordered in the order observed by correct processes. The key idea is simply to piggyback timing information in the messages of the classical reliable broadcast [113], hence preserving the time and message complexities of the original algorithm. This protocol is interesting in its own right to mitigate front-running attacks in broadcast-based cryptocurrencies [43].

2. We devise a fast ordered Byzantine consensus protocol that addresses the limitations of the Ordered Reliable Broadcast protocol. This ordered consensus protocol consists of an ordered reliable broadcast followed by a series of binary consensus instances to decide whether we can order each transaction within a given time window. If the consensus for transaction t in the τ^{th} time window returns 1, then τ becomes the timestamp of transaction t. By parallelising the steps of ordering and agreeing on a block, the fast ordered Byzantine consensus protocol takes 6 message delays to terminate in normal executions, almost twice faster than the original protocol [13].

3. We build upon these protocols to implement a blockchain that ensures ordering linearizability. It builds upon our newly devised consensus algorithm and benefits from the recent advances in blockchain [74] by committing more than one proposal per consensus instance and by reducing the verification overhead. We deploy it in a worldwide environment of forty machines where it can commit tens of thousands of ordered transactions per second. Finally, we compare its performance to a recent scalable blockchain, Red Belly Blockchain [74]. Although slower than Red Belly Blockchain, our blockchain demonstrates that one can devise a practical blockchain that mitigates front-running attacks.

Chapter Outline. This chapter is organised as follows. We define the type of ordering indicators used in this chapter in Section 3.1. The Ordered Reliable Broadcast protocol is introduced in Section 3.2. The Ordered Consensus protocol is detailed in Section 3.3. In Section 3.4, we apply these algorithms to a blockchain, and in Section 3.5, we evaluate this blockchain. We summarise and conclude the chapter in Section 3.6.

3.1 Local Clocks and Timestamps

In this chapter, we assume that processes use the Clock implementation of the Rank function, i.e., Rank = Clock (see Section 2.2.6). Each process p_i has a local clock Clock_i that is uses to determine the *timestamps* that it assigns to transactions:

$$\mathsf{Clock} : \mathcal{T} \longrightarrow \mathbb{N},$$
$$t \longmapsto s = \mathsf{Clock}_i(t).$$

3.2 Ordered Reliable Broadcast

In this section, we present the Ordered Reliable Broadcast protocol as a variant of reliable broadcast [113]. An ordered reliable broadcast delivers each transaction with a set S of times-

tamps. The median value of the set S is a timestamp that is correctly bounded. This ordered reliable broadcast serves as a building block to our Ordered Consensus protocol (Section 3.3) that is itself key to our blockchain implementation (Section 3.4).

3.2.1 Piggybacking Local Clock Values

Reliable broadcast [113] is a broadcast protocol that ensures that if a message is broadcast by a correct process, it will eventually be delivered by all correct processes. To achieve an ordering property on delivered transactions, our ordered reliable broadcast variant piggybacks a signed timestamp on ECHO messages and a set of $2f + 1$ signed timestamps on READY messages. As a result, it does not introduce any message delays to the original reliable broadcast. Each transaction that is reliably delivered comes with a set of $2f + 1$ timestamps whose median value can be used as a sequence number to locally sequence the transaction. The complete protocol is detailed in Algorithm 3.1. The ordered reliable broadcast consists of the same three steps as its classical counterpart:

1. INIT: process p_i broadcasts its transaction t to all the processes (line 2);
2. ECHO: every process that receives an INIT message with a transaction t from a process p_j broadcasts an ECHO message containing t and the signed value of its local clock (line 6);
3. READY: when a process p_i receives $n - f$ ECHO messages with the transaction t, it broadcasts a READY message with t and the set S of $2f + 1$ signed timestamps received in ECHO messages (line 9); if a process receives $f + 1$ READY messages, it can directly broadcast a READY message with a set S containing the $2f + 1$ lowest timestamps received in the $f + 1$ READY messages (line 13);

Whenever a process receives $n - f$ READY messages with the transaction t, it delivers t and a set containing the $2f + 1$ lowest timestamps that it has received in the previous steps (line 17).

3.2.2 Correctness of the Ordered Reliable Broadcast

In this section, we show that the Ordered Reliable Broadcast protocol preserves the properties of the reliable broadcast and satisfies ordering linearizability.

Theorem 3.1 (Reliable Broadcast)**.** *The ordered reliable broadcast (cf. Algorithm 3.1) preserves the properties of the reliable broadcast protocol, that is:*

- *if a correct process broadcasts a transaction t, then t is eventually delivered to all correct processes;*
- *if a faulty process broadcasts a transaction t, then either t is eventually delivered to all correct processes or t is never delivered to any correct process.*

Proof. If a process, correct or Byzantine, broadcasts a transaction t, our variant only piggybacks a signed timestamp on ECHO messages and a set of signed timestamps on READY messages. The

Algorithm 3.1 Ordered Reliable Broadcast

1: **function** OR-Broadcast(t) ▷ *ordered variant of reliable broadcast at p_i*
2: Broadcast(INIT, t) ▷ *broadcast transaction t*
3: **upon** receiving a message (INIT, t) from p_j **do**
4: $s \leftarrow \mathsf{Clock}_i(t)$ ▷ *assign timestamp to t*
5: $\sigma \leftarrow \mathsf{DS.Sign}(sk_i, s)$ ▷ *sign timestamp*
6: Broadcast(ECHO, (t, s, σ, i)) ▷ *echo t and assigned timestamp*
7: **upon** receiving $n - f$ (ECHO, (t, s, σ, j)) messages and not having sent READY **do**
8: $S \leftarrow \bigcup_{2f+1}((s, \sigma))$ ▷ *build set of timestamps*
9: Broadcast(READY, (t, S, i))
10: **upon** receiving $f + 1$ (READY, (t, S, j)) messages and did not send READY **do**
11: $S_{all} \leftarrow \bigcup_{\forall S, \forall (s,\sigma) \in S}((s, \sigma))$ ▷ *set of all received timestamps*
12: $S \leftarrow S_{all}[1..2f + 1]$ ▷ *lowest $2f + 1$ timestamps*
13: Broadcast(READY, (t, S, i))
14: **upon** receiving $n - f$ (READY, (t, S, j)) messages and not delivered from p_j **do**
15: $S_{all} \leftarrow \bigcup_{\forall S, \forall (s,\sigma) \in S}((s, \sigma))$ ▷ *set of all received timestamps*
16: $S \leftarrow S_{all}[1..2f + 1]$ ▷ *lowest $2f + 1$ timestamps*
17: Deliver(t, S) ▷ *deliver t and set S of timestamps*

set of $2f + 1$ timestamps can always be built because a READY message is only broadcast by correct processes after either receiving $n-f \geq 2f+1$ ECHO messages, where each ECHO message contains a signed timestamp, or receiving $f + 1$ READY messages, where each READY message contains a set of $2f + 1$ signed timestamps. These added metadata are delivered alongside t and never prevent the delivery of t. □

Theorem 3.2 (Ordering Linearizability)**.** *The ordered reliable broadcast (cf. Algorithm 3.1) satisfies ordering linearizability. Formally,*

$$\forall p_i \in P^{Corr}, \forall t_1, t_2 \in \mathsf{Log}_i, \mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2) \Rightarrow t_1 \prec t_2.$$

Proof. Transactions delivered by the ordered reliable broadcast are ordered using the median value of the set of $2f + 1$ timestamps delivered with each transaction. Because the median timestamp is greater than or equal to f values, it is at least equal to a timestamp perceived by a correct process. In the same way, because it is upper bounded by at least f values, it is also less than or equal to a timestamp perceived by a correct process. The median timestamp computed for each transaction t is thus upper-bounded and lower-bounded by the timestamps perceived by correct processes. As a result, if all the timestamps perceived by correct processes for a transaction t_1 are lower than any timestamp perceived by correct processes for a transaction t_2, then for all correct processes, the median timestamp computed for t_1 will be lower than the median timestamp of t_2, and thus $t_1 \prec t_2$. □

It is interesting to note that there is no guarantee that the median timestamp associated with a transaction will be identical across different processes. This is due to Byzantine processes

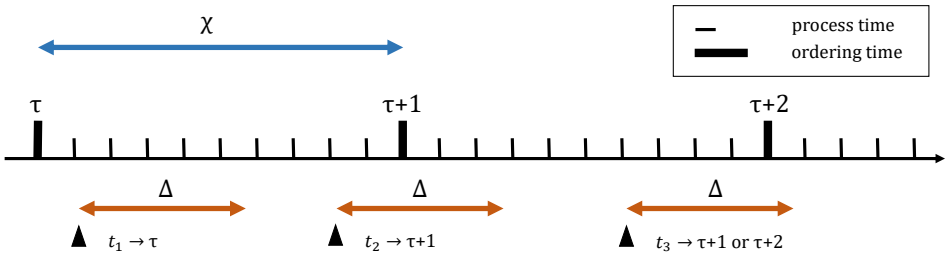

Figure 3.1: Transaction t_1 will be sequenced with the ordering indicator τ, while transaction t_2 will likely be sequenced with the ordering indicator $\tau + 1$. Because some correct processes may observe transaction t_3 in $\tau + 1$ while others in $\tau + 2$, the ordering indicators computed by correct processes may vary for t_3.

sending different timestamps to different processes instead of broadcasting the same timestamp. This different distribution results in correct processes selecting a different value as the median of their distribution. In the following, we show how to cope with this problem by solving the ordered Byzantine consensus problem.

3.3 Fast Ordered Byzantine Consensus

3.3.1 Logical Time

Each transaction that is delivered by the ordered reliable broadcast comes with (i) a *reference timestamp* θ provided as metadata of the transaction by the proposer, and (ii) a set S of $2f + 1$ signed timestamps gathered during the ordered reliable broadcast. Due to the fact that correct processes may not have gathered the same timestamps, or because of the effect of Byzantine processes, the set S and thus its median value are not equal at each process. In order to obtain a total order on all transactions [114] and implement an SMR, processes need to reach an agreement on a unique *ordering indicator* to adopt for each transaction. To simplify the agreement process, we introduce an ordering clock with a coarser grain than the process clock. Instead of agreeing on a timestamp coming from a process clock, processes will agree on an ordering indicator coming from the ordering clock. The idea is to decide as *ordering indicator* a value of the ordering clock that results from a timestamp that is correctly bounded.

Ordering clock

Let $\chi > 0$ be a global constant. Each unit of the ordering clock lasts χ units of the process clock. Each timestamp s coming from a process clock can be mapped to an ordering indicator $\mathsf{Order}(s) = \tau$ on the ordering clock, with $\tau\chi \leq s < (\tau + 1)\chi$. χ is the *granularity* with which transactions are ordered by each process. Figure 3.1 shows examples of how processes may adopt a value on the ordering clock.

Correctly bounded ordering indicators

By extension to Definition 2.4, we say that an ordering indicator τ associated to a transaction t is *correctly bounded* if it results from a timestamp s associated with t that is correctly bounded:

$$\tau \text{ is correctly bounded} \Leftrightarrow \exists s \mid (s \text{ is correctly bounded} \wedge \mathsf{Order}(s) = \tau).$$

3.3.2 Order Agreement

The goal of the agreement procedure is to decide an ordering indicator τ_t on the ordering clock for a transaction t. This is achieved in three steps:

1. First, each process computes the ordering indicator τ associated to the reference timestamp θ provided by the proposer using $\tau = \mathsf{Order}(\theta)$. Because of the properties of the reliable broadcast, the reference timestamp is identical at each correct process because it is part of the transaction.

2. Then, each process adopts an ordering indicator estimate e based on the set of timestamps delivered by the ordered reliable broadcast. This estimate is the ordering indicator of the median value $\overline{s}$ of the set of $2f + 1$ timestamps delivered during the ordered reliable broadcast, i.e., $e = \mathsf{Order}(\overline{s})$.

3. Finally, processes start a procedure where they share their estimates and agree on a unique ordering indicator that is correctly bounded.

In order to agree on a unique ordering indicator for a transaction, processes use the ordering indicator τ of the reference timestamp of the transaction as a reference point and try to agree on an ordering indicator $\tau_t = \tau + k$. The order agreement protocol can thus be reduced to deciding a value k to be added to the reference point τ. To achieve this, processes execute successive rounds of binary consensus on the current round value, starting with the round $k = 0$ (i.e., $\tau_t = \tau + 0$) and increasing the round number by one if the consensus outputs 0. If the consensus instance associated with round k outputs 1, then the ordering indicator $\tau_t = \tau + k$ is adopted for the transaction. The reason for this approach is that after GST, and provided that the value of χ is greater than Δ, a transaction is received by correct processes within a delay χ. As a result, a transaction is broadcast and received either during the same unit of ordering time (i.e., at $\tau + 0$) or during the next one (i.e., at $\tau + 1$), and the ordering indicator is decided during one of the first two instances of binary consensus (i.e., $k = 0$ or $k = 1$). Otherwise, processes may require several successive instances of consensus to decide the ordering indicator. To reduce the number of message delays in synchronous cases, we execute the first two rounds concurrently.

The algorithm for deciding the value of k is presented in Algorithm 3.2. First, each process adopts as its local estimate the ordering indicator of the median value of the set S of timestamps delivered by the ordered reliable broadcast (line 20). Before trying all possible values of k, processes first try to decide the values 0 and 1 concurrently within a fast path (line 21). Then, during each iteration of the loop, a process starts by checking if the current round number can

Algorithm 3.2 Order Agreement

18: **function** OrderedAgreement(t, S)
19: $\theta \leftarrow t.timestamp$ ▷ *reference timestamp of the proposal*
20: e $\leftarrow$ Order(Median(S)) ▷ *local estimate of ordering indicator*
21: **Execute concurrently**
22: • OrderPropose$(0, \text{e} = \text{Order}(\theta), \theta, S) \rightarrow zeroDecided$
23: • OrderPropose$(1, \text{e} = \text{Order}(\theta) + 1, \theta, S) \rightarrow oneDecided$
24: **if** $zeroDecided = 1$ **then**
25: **return** 0 ▷ *decide 0 as ordering indicator*
26: **else if** $oneDecided = 1$ **then**
27: **return** 1 ▷ *decide 1 as ordering indicator*
28: $r \leftarrow 2$ ▷ *start with round 2*
29: **loop**
30: **if** $e = \text{Order}(\theta) + r$ **then**
31: $vote \leftarrow 1$ ▷ *vote 1 if p_i's estimate is r*
32: **else**
33: $vote \leftarrow 0$
34: OrderPropose$(r, vote, \theta, S) \rightarrow rDecided$ ▷ *consensus to adopt r as ordering indicator*
35: **if** $rDecided = 1$ **then**
36: **return** r ▷ *ordering indicator r decided*
37: OrderBacktrack$(r, \theta, S) \rightarrow backtrackOrder$ ▷ *try a lower ordering indicator*
38: **if** $backtrackOrder \neq \perp$ **then**
39: **return** $backtrackOrder$ ▷ *backtrack decided*
40: $r \leftarrow r + 1$ ▷ *increment the round number*
41: **end loop**

be adopted (line 30), i.e., if the current round number r would result in an ordering indicator $\tau + r$ that is correctly bounded. If that is the case, the process votes 1 to the consensus instance associated with the current round number (line 34).

If the order agreement outputs 1, then the current round number is decided, and $\tau_t = \tau + r$; otherwise, a backtracking step is performed (line 37). Backtracking (see Section 3.3.5) enables processes to adopt an ordering indicator broadcast by a rotating coordinator. The backtracking mechanism has been introduced because a network adversary could prevent correct processes from reaching an agreement until the round number goes beyond a value that would result in an ordering indicator that is correctly bounded.

3.3.3 Binary Consensus on Orders

The OrderPropose algorithm for binary consensus on each round number is detailed in Algorithm 3.3 and is a slight variation of the DBFT algorithm [33]. The method OrderPropose takes four arguments:

Algorithm 3.3 Binary Consensus on Order

42: **function** OrderPropose(k, val, θ, S) ▷ *binary consensus at p_i with $val \in \{0, 1\}$*
43: **loop** ▷ *loop that starts with round $r = 1$*
44: OV-Broadcast(val, θ, k, S) $\rightarrow$ *cvals* ▷ *order value broadcast*
45: StartTimer(r) ▷ *timeout increases with rounds*
46: **if** $i = r$ mod n **then** ▷ *coordinator rebroadcasts*
47: **wait until** ($cvals = \{w\}$) ▷ *cvals stores delivered values*
48: Broadcast(COORD, r, w) $\rightarrow c$ ▷ *coordinator broadcasts*
49: **wait until** ($cvals \neq \emptyset \wedge$ timer expired) ▷ *wait enough time*
50: **if** $c \in cvals$ **then** $e \leftarrow \{c\}$ **else** $e \leftarrow cvals$ ▷ *prioritise coord value*
51: Broadcast(AUX, r, e) $\rightarrow$ *bvals* ▷ *broadcast these values*
52: **wait until** $\exists b \subseteq bvals$ where the two following conditions hold:
53: • b contains contents received from at least $n - f$ distinct processes
54: • $\forall v \in b, v \in cvals$ ▷ *every value in b is in cvals*
55: **if** $b = \{v\}$ **then** ▷ *if there is only one value in b*
56: $val \leftarrow v$ ▷ *adopt this singleton value*
57: **if** $v = (r$ mod $2)$ and not decided yet **then** Decide(v) ▷ *decide*
58: **else** $val \leftarrow (r$ mod $2)$ ▷ *otherwise, adopt the current parity bit*
59: **if** decided in round $r - 2$ **then** Exit() ▷ *help others in two last rounds*
60: $r \leftarrow r + 1$ ▷ *increment the round number*
61: **end loop**

- k: the value of the round that is being decided; this value is used by the underlying OV-Broadcast protocol;

- val: the value input by the process to the binary consensus, a process will input 1 to a consensus instance if its local estimate corresponds to the current round number, 0 otherwise;

- θ: the value of the transaction's reference timestamp; this value is used by the underlying OV-Broadcast protocol;

- S: a set of timestamps to support a value $val = 1$; it is used as a proof in OV-Broadcast so that other correct processes rebroadcast 1 if the median value of the set S is correctly bounded.

This protocol allows correct processes to decide on an ordering indicator that is correctly bounded. The DBFT protocol relies on the BV-Broadcast protocol [31] for the reliable broadcast of binary values. We modified the BV-Broadcast protocol to obtain the OV-Broadcast protocol (detailed in Section 3.3.4).

Algorithm 3.4 Order Value Broadcast

62: **function** OV-Broadcast(k, v, θ, S) ▷ *order value broadcast at p_i*
63: Broadcast$(\text{EST}, k, v, \theta, S)$ ▷ *broadcast v, reference timestamp θ, round k, and S*
64: **upon** receiving a message $(\text{EST}, k, 1, \theta, S)$ from p_j **do**
65: **if** Order(Median(S)) = Order$(\theta) + k$ **then** ▷ *check if correctly bounded*
66: Broadcast$(\text{EST}, k, 1, \theta, S)$ ▷ *rebroadcast 1 if not already done*
67: **upon** receiving $f + 1$ messages $(\text{EST}, k, v, \cdot)$ **do**
68: Broadcast$(\text{EST}, k, v, \cdot)$ ▷ *rebroadcast v if not already done*
69: **upon** receiving $n - f$ $(\text{EST}, k, v, \cdot)$ and v not delivered **do**
70: OV-Deliver(v)

3.3.4 Order Value Broadcast

The OV-Broadcast protocol enables broadcasting binary values associated with an ordering indicator. In the initial BV-Broadcast protocol, a value is rebroadcast by a correct process if it has been received from at least $f + 1$ processes, i.e., from at least one correct process. This ensures that if a value is delivered by the protocol, then it has been broadcast by at least one correct process. Our OV-Broadcast variant adds another scenario where a correct process can rebroadcast the value 1. A correct process rebroadcasts the binary value 1 when it receives it during round k from at least one process, and the ordering indicator resulting from the selection of round k is correctly bounded (line 65). The goal of the OV-Broadcast protocol is to enable the broadcast of a value while guaranteeing non-intrusion [115]. Non-intrusion means that the protocol filters out the values coming exclusively from Byzantine processes. In our case, it means preventing the broadcast of a value that is not correctly bounded by relying on a set of $2f + 1$ signed timestamps. The OV-Broadcast protocol is presented in Algorithm 3.4. It enables a correct process to broadcast the value 1 associated with an ordering indicator that is correctly bounded. At line 66, a process rebroadcasts the value 1 for a round k if it is justified by a set S of signed timestamps, and if deciding 1 for round k would result in a correctly bounded ordering indicator.

3.3.5 Backtracking

During the backtracking step, any ordering indicator that would result in an ordering indicator less than the current round can be decided, assuming that a proof is provided by the coordinator. The backtracking mechanism is presented in Algorithm 3.5. If the ordering indicator of the current round is not decided, the backtracking step occurs (line 37), and a rotating coordinator is allowed to propose a set of timestamps that would result in the adoption of a correctly bounded ordering indicator. The reliable broadcast [113] (line 75) ensures that the same set of timestamps is delivered to all correct processes. Processes then perform a round of binary consensus [33] to decide whether to adopt the ordering indicator received from the coordinator. Each process verifies the set of timestamps received from the coordinator (line 78) and votes 1 to the binary consensus if the verifications pass, 0 otherwise.

Algorithm 3.5 Order Backtrack

```
71: function OrderBacktrack(r, θ, S)                          ▷ backtrack consensus at p_i
72:     if i = (r mod n) then                                  ▷ p_i is coordinator
73:         if Order(Median(S)) − Order(θ) < r then            ▷ check value from coordinator
74:             k ← Order(Median(S)) − Order(θ)                ▷ compute k
75:             ReliableBroadcast((k, S)) → cvals              ▷ broadcast k and proof
76:     StartTimer(r)                                          ▷ increase timer with round number
77:     wait until (cvals ≠ ∅ ∨ timer expired)
78:     if (∃(k, S) ∈ cvals) ∧ (k < r) ∧ (Order(Median(S)) = Order(θ) + k) then
79:         BinPropose(1) → backtrackDecided                   ▷ input 1 to binary consensus
80:     else
81:         BinPropose(0) → backtrackDecided                   ▷ input 0 to binary consensus
82:     if backtrackDecided then
83:         return k                                           ▷ coordinator order was decided
84:     else
85:         return ⊥
```

3.3.6 Correctness of the Fast Ordered Byzantine Consensus

The fast ordered Byzantine consensus enables correct processes to decide a unique ordering indicator that is correctly bounded for each transaction.

Theorem 3.3 (Order Value Broadcast)**.** *The order value broadcast (cf. Algorithm 3.4) provides the same properties as the binary value broadcast.*

- ***Termination.*** *Invocation by a correct process terminates.*
- ***Justification.*** *If a value v is delivered at a correct process, then either it has been broadcast by a correct process, or the resulting ordering indicator is correctly bounded.*
- ***Uniformity.*** *If a value is delivered at a correct process, then it is eventually delivered at all correct processes.*
- ***Obligation.*** *Eventually, all the sets cvals of all correct processes become not empty.*

Proof. The proof is the same as the proof for the BV-Broadcast algorithm [31], except for *Justification*, where the definition of validity has been extended.

Termination. When a process OV-Broadcasts its input value, it eventually finishes sending a message to each process.

Justification. If a value is broadcast only by Byzantine processes, then correct processes will not rebroadcast this value because they receive it at most f times, with $f < \frac{n}{3}$, and the condition on line 67 is never satisfied. Because Byzantine processes are computationally bounded, they cannot forge signed timestamps and build a set of $2f + 1$ values that justifies an ordering indicator that is not correctly bounded. As a consequence, the predicate on line 65

cannot be satisfied. Because the $n - f$ correct processes only rebroadcast values that either result in ordering indicators that are correctly bounded or have been received from at least $f + 1$ processes, other values are never delivered in the set *cvals* of correct processes.

Uniformity. If a process adds a value v to its set *cvals*, then it received it from $2f + 1$ processes, out of which at least $f + 1$ are correct. These $f + 1$ correct processes broadcast v to all processes, and as a result, all correct processes will receive at least $f + 1$ messages and rebroadcast v. Eventually, all correct processes will have broadcast v, and all correct processes will receive at least $n - f \geq 2f + 1$ messages with v and deliver v.

Obligation. Every time OV-Broadcast is called, for each consensus instance on orders, it is called by every correct process. Because there are at least $n - f \geq 2f + 1$ correct processes, and that $2f + 1 = (f + 1) + f$, at least one of the binary values will be broadcast by at least $f + 1$ correct processes and thus be rebroadcast and eventually delivered by all correct processes. □

Theorem 3.4 (Binary Consensus)**.** *The* OrderPropose *algorithm (cf. Algorithm 3.3) ensures* validity*,* agreement*, and* termination*.*

Proof. Our variant of the DBFT algorithm [33] relies on a modification of the OV-Broadcast algorithm so that the validity property now encloses values broadcast by correct processes and values that result in ordering indicators that are correctly bounded. As a direct consequence, because we do not modify the consensus algorithm, and because of Theorem 3.3, the validity, agreement, and termination properties of binary consensus are preserved. □

Theorem 3.5 (Order Agreement)**.** *The* OrderedAgreement *algorithm (cf. Algorithm 3.2) ensures that all correct processes agree on the same ordering indicator for each transaction.*

Proof. When a transaction t is delivered by the ordered reliable broadcast, each correct process starts an instance of OrderedAgreement to decide an ordering indicator for t. We prove that at every correct process, this operation terminates (**termination**), that the ordering indicator agreed upon is the same at each correct process (**agreement**) and that it is correctly bounded (**validity**).

Validity. The validity property comes directly from the validity properties of the abstractions used.

Agreement. Whenever a process stops the OrderedAgreement procedure because it has decided, it is either after an outcome of 1 in one of the consensus instances of lines 22, 23, or 34, or after a backtracking decision from line 37. Because of the agreement property of the OrderPropose algorithm and the fact that correct processes wait for the result of an agreement protocol before moving on to the next step, when a value 1 is decided, all correct processes will exit at the same round and thus decide the same round value either in line 25, 27, or 36. If the value is decided during a backtrack, then during the backtrack, the value 1 was proposed to the binary consensus (line 79) by at least $f + 1$ processes, and thus by at least one correct process. This correct process has received the set of timestamps broadcast by the coordinator via a reliable broadcast, and therefore, the same set of timestamps is eventually delivered to all the correct processes. Because of the agreement property of the binary consensus used during the backtracking step, all correct processes either decide 0 and return $\perp$ or decide 1 associated with

the unique ordering indicator resulting from the delivered set of timestamps.

Termination. During the ordered reliable broadcast, a correct process delivers a transaction after receiving $n-f$ READY messages, and each message contains a set of timestamps. A process delivers with a transaction the $2f+1$ lowest timestamps received in the READY messages. At each correct process, the median value of this set of timestamps is used as the initial estimate of the ordering indicator of the transaction.

Let K_t be the set of estimates adopted by the correct processes for a transaction t, and let $k_{min} = \min_{\forall k \in K_t}(k)$ and $k_{max} = \max_{\forall k \in K_t}(k)$. Suppose the network behaves synchronously during any of the rounds k_{min} to k_{max}. Without loss of generality, suppose that the round $k_{syn} \in K_t$ behaves synchronously and that the value k_{syn} was adopted by some process p_{syn}. During round k_{syn}, process p_{syn} inputs 1 to the OrderPropose protocol at line 34, and thus inputs the value 1 and its set of timestamps to the OV-Broadcast protocol at line 44. Due to the synchronous behaviour of the round and to the validity property of OV-Broadcast, the input from p_{syn} is delivered to correct processes in a timely manner, and all correct processes have at least the value 1 in their sets *cvals* and bvals. As a result, all correct processes adopt 1 as their estimates, either because they only have the value 1 in bvals, or because they adopt the parity of the round, and thus the value k_{syn} is decided, terminating the consensus.

On the other hand, if no round corresponding to a value in K_t behaves synchronously between k_{min} and k_{max}, we need to prove that the backtracking algorithm eventually terminates because the round number cannot be decided anymore, as it would result in an ordering indicator that would not be correctly bounded. During the backtracking step, first, a coordinator submits its set of timestamps to a reliable broadcast, then each process waits (line 77) until either a value from the coordinator has been received or the timeout has expired. If a valid set of timestamps is received, a process proposes 1 to the backtracking consensus associated with the delivered values, whereas if an invalid set or no set has been received, the timeout will expire, and the process proposes 0. Because the timeout for receiving the set from the coordinator increases with each round, there is a round k_{syn} after which the network starts behaving synchronously. From there on, and because the coordinator changes at each round, a correct process will eventually be selected as coordinator and have its set of timestamps delivered in time to correct processes, who in turn can decide the associated ordering indicator.

□

Theorem 3.6 (Ordering Linearizability)**.** *The ordered Byzantine consensus (cf. Algorithm 3.2) ensures ordering linearizability with respect to the ordering clock. Let S_1 (resp. S_2) denote the set of timestamps assigned by correct processes to transaction t_1 (resp. t_2). Then,*

$$\forall s_1 \in S_1, s_2 \in S_2, \mathsf{Order}(s_1) < \mathsf{Order}(s_2) \Rightarrow t_1 \prec t_2.$$

Proof. Assume that S_1 and S_2 are the sets of timestamps assigned by correct processes to two transactions t_1 and t_2, respectively. We denote by $\mathsf{MaxOrder}(t_1) = \max_{\forall s \in S_1}(\mathsf{Order}(s))$ the highest ordering indicator of t_1 that is correctly bounded, and by $\mathsf{MinOrder}(t_2) = \min_{\forall s \in S_2}(\mathsf{Order}(s))$ the lowest ordering indicator for t_2 that is correctly bounded. From the hypothesis, we have

$$\forall s_1 \in S_1, s_2 \in S_2, \mathsf{Order}(s_1) < \mathsf{Order}(s_2) \Rightarrow \mathsf{MaxOrder}(t_1) < \mathsf{MinOrder}(t_2).$$

Let o_1 be the ordering indicator decided for transaction t_1, and o_2 the ordering indicator decided for t_2. Due to the validity property of the ordered Byzantine consensus, both o_1 and o_2 are correctly bounded. Therefore, we have

$$o_1 \leq \mathsf{MaxOrder}(t_1) \wedge o_2 \geq \mathsf{MinOrder}(t_2) \Rightarrow o_1 < o_2.$$

Using the agreement property of the ordered Byzantine consensus, we can conclude that $t_1 \prec t_2$.

□

3.3.7 Time Complexity

In this section, we analyze how the value of χ impacts the order agreement algorithm by studying the ratio $\alpha = (\Delta + \delta)/\chi$, where δ is the maximum offset between the values of two clocks. When $\alpha < 1$, all correct processes receive transactions within a delay χ, and adopt as estimates either $e = 0$, if the transaction was broadcast and received inside the same unit of the ordering clock, or $e = 1$, when the transaction was broadcast towards the end of a unit and received by processes during the next unit. In these conditions, when the network is synchronous, the ordering indicator is decided in the concurrent executions of the order agreement for the values $r = 0$ and $r = 1$. This results in the ordering indicator of the transaction being decided after 6 message delays: 3 message delays for the three steps of the ordered reliable broadcast and 3 message delays for the binary consensus. Nevertheless, a value of χ that is large will result in lesser precision in the ordering, as succeeding transactions risk having the same ordering indicator.

Inversely, if $\alpha \geq 1$, the ordering becomes more precise, but each transaction may be received by processes on potentially different values of the ordering clock, and each process may adopt a different ordering indicator estimate. In this case, the number of consensus instances on ordering indicators is proportional to α. During each round of the order agreement algorithm, processes first try to decide the current round number and then try backtracking. While executing these two steps, any ordering indicator that is correctly bounded can be decided. This can happen either during the consensus phase when adopting the round number, provided that a process broadcasts a set of timestamps satisfying the round number, or during the backtracking step, if the set of timestamps from the coordinator is broadcast in a timely manner.

3.4 Application to Blockchains

In this section, we apply the fast ordered Byzantine consensus protocol of Section 3.3 to the context of blockchains. To this end, we consider the open model [32] where permissionless processes called *clients* can issue transactions, but only n permissioned processes called *replicas* can participate in the consensus protocol to order these transactions. More precisely, as in recent open blockchains [74], clients submit their transactions to replicas until they receive acknowledgements from $f + 1$ distinct replicas that their transactions are valid, where $f < n/3$ replicas can be Byzantine. This ensures that at least one correct replica stores every valid

transaction in its memory pool (or *mempool*). Each correct replica batches transactions from its mempool into a block proposal and proposes it in the next consensus instance. Other replicas participate in the next consensus instance by receiving proposals or proposing themselves. Once a consensus on a set of proposed transactions is reached, this set is recorded to reliable storage and executed at each correct replica.

3.4.1 Block Agreement and Reconciliation

The block-building process is described in Algorithm 3.6 and borrows the superblock optimisation introduced in [33]. The superblock optimisation aims to improve throughput and scalability. To this end, it leverages set Byzantine consensus whereby processes decide a set of values. Hence, each decided block combines proposals from a set of processes.

A process starts by broadcasting its proposal to other processes (line 87) and then takes part in the consensus instances for all the proposals that it has received. The first consensus (line 91) is a Byzantine binary consensus that decides if the proposal is included in the block, and the second consensus (line 92) is an ordered agreement that determines the ordering indicator to adopt for the proposal. Once the decided block contains at least $n - f$ proposals (line 89), processes propose 0 to binary consensus instances associated with the proposals that have not been delivered (line 94). After all the binary consensus instances have terminated, a process waits until it has received all the decided proposals and all the ordering indicators associated with these proposals have been decided (line 95) and proceeds with reconciliation.

More precisely, a replica broadcasts a proposed batch of transactions as soon as the first of these two predicates holds: either its mempool contains a threshold (1000 by default) of transactions or a certain amount of time has elapsed since its last proposal. The difference between our implementation and previous open blockchains that combine reliable broadcast with binary consensus [32, 74] lies in two things. First, the replica broadcasts a batch using our ordered reliable broadcast (Section 3.2) instead of the classical reliable broadcast. Second, whenever a replica receives a proposal from the ordered reliable broadcast, it starts *two* concurrent consensus instances instead of a single consensus instance. These two concurrent consensus instances help decide (*i*) whether the proposal will be included in the next block and (*ii*) its ordering indicator. Following the superblock optimization [74], once the proposal of each replica is decided, replicas execute a reconciliation process where block proposals are combined into a superblock. During this reconciliation, all the non-conflicting transactions output by the superblock are executed according to their ordering indicators or according to a lexicographical order in case of a tie. The reconciliation process is presented in Algorithm 3.7.

Properties

Similar to previous work [74], our blockchain implements the set Byzantine consensus problem (cf. Theorem 3.7), a variant of the original Byzantine consensus problem that offers high performance at a large scale. To this end, it ensures that each decided block is made of a union of valid proposals.

Algorithm 3.6 Bloc Agreement

86: **function** BatchPropose(v) ▷ *set Byzantine consensus at p_i with v a batch of txs*
87: OR-Broadcast(v) $\rightarrow$ *props*
88: StartTimer(age of oldest tx in mempool) ▷ *give time to slow ones*
89: **while** $|\{k : bitmask[k] = 1\}| < n - f$ or *timer* did not expire **do**
90: **for all** k such that *props*$[k]$ has been delivered ▷ *for all delivered*
91: $bitmask[k] \leftarrow \mathsf{BinPropose}_k(1)$ ▷ *propose 1 to k^{th} binary consensus*
92: $order[k] \leftarrow \mathsf{OrderedAgreement}_k(props[k])$ ▷ *propose S to k^{th} order agreement*
93: **for all** k such that *props*$[k]$ has not been delivered ▷ *for undelivered*
94: $bitmask[k] \leftarrow \mathsf{BinPropose}_k(0)$ ▷ *propose 0 to k^{th} binary consensus*
95: **wait until** *bitmask* is full and $\forall \ell, bitmask[\ell] = 1 : props[\ell] \neq \emptyset \wedge order[\ell] \neq \perp$
96: Reconciliate($bitmask$ & $props, bitmask$ & $order$) ▷ *combine into a superblock*

Algorithm 3.7 Reconciliation

97: **function** Reconciliate($props, order$) ▷ *combine proposals, initially superblock = $\emptyset$*
98: $props \leftarrow \mathsf{Sort}(props, order)$ ▷ *sort proposals according to their decided timestamps*
99: **for** $i = 0..(n-1)$ **do**
100: **for** $tx \in props[(i)]$ **do**
101: **for** $ctx \in superblock$ **do** ▷ *for all committed transactions*
102: **if** $\neg\mathsf{Conflict}(tx, ctx)$ **then** $superblock \leftarrow superblock \cup \{tx\}$
103: Decide($superblock$) ▷ *decide superblock of non-conflicting transactions*

Theorem 3.7 (Set Byzantine Consensus)**.** *If all correct replicas propose a set of transactions, each of the following properties holds.*

- ***Termination.*** *A correct replica eventually decides.*
- ***Agreement.*** *If two correct replicas decide block b and block b', then $b = b'$.*
- ***Validity.*** *The decided block contains only valid and non-conflicting transactions, and the decided set of transactions is a subset of the union of the proposed sets.*
- ***Non-triviality.*** *If all replicas are correct and they all propose the same set of transactions, then the decided block contains exactly this set of transactions.*

Proof. We prove each property separately.
Termination. Termination follows directly from the termination properties of the Byzantine binary consensus and ordered agreement protocols.
Agreement. Because both consensus on block inclusion and orders ensure agreement, all correct processes decide the same set of proposals and associated ordering indicators. As a result, they obtain the same set of transactions in the same order during reconciliation.
Validity. Validity is guaranteed by the validity properties of the underlying consensus abstractions and the fact that each correct process deterministically verifies if a transaction is in

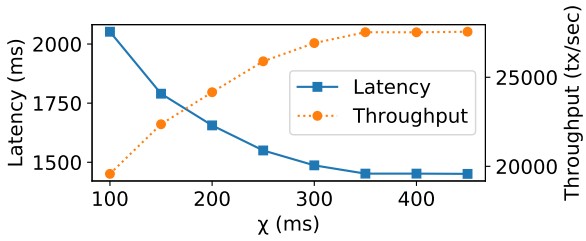

Figure 3.2: Using a single datacenter, performance stabilises for values of χ greater than 350 ms.

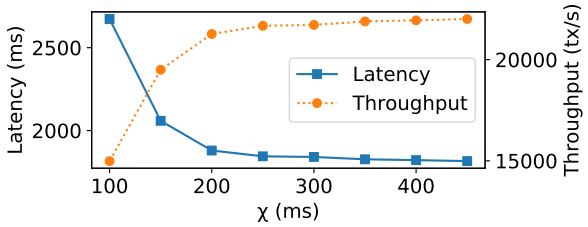

Figure 3.3: With multiple datacenters, performance becomes stable for a value of χ around 450 ms.

conflict before adding it at line 102.

Non-triviality. The decided set results from the union of $n - f$ proposals. If all processes are correct and propose the same proposal, then this proposal is necessarily contained in the decided set. □

Each decided block is appended to the existing chain of previously decided blocks, thus resulting in a distributed ledger of totally ordered transactions.

3.5 Experimental Evaluation

In this section, we present our experimental results. We use the ordered consensus algorithm to sequence the proposals in each block of our blockchain, and evaluate its scalability by measuring the throughput and the latency when increasing the number of processes. These experiments are conducted first within a single datacenter and then on different continents in 5 AWS regions (Frankfurt, Ireland, Ohio, North California and Oregon). We then compare our results to the Red Belly Blockchain, which does not satisfy ordering linearizability. Note that we do not compare our results with Pompē because the Pompē experiments reach consensus on the hash values of transactions, and the transmission of payloads is done outside the protocol. In our case, clients submit real transactions whose validity is verified by replicas. Each process (acting both as a client and a replica) is a *c4.4xlarge* EC2 virtual machine (16 vCPUS, 30 GB of RAM), and each proposal contains 1000 transactions.

Measuring χ

The fast path of the algorithm requires that the decided ordering indicator for each proposal is either 0 or 1. We measure the latency and throughput of the blockchain with 40 processes using various values of χ. Figure 3.2 shows the results when all 40 processes are located in the same datacenter. In Figure 3.3, the 40 processes are equally distributed across 5 datacenters in distinct regions.

When the value of χ is less than the message propagation time Δ, the agreement on the ordering indicator of a transaction requires successive instances of binary consensus that decrease the performance of the system. By contrast, when χ is greater than Δ, all proposals are decided

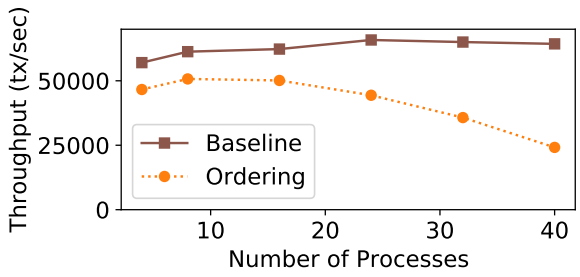

Figure 3.4: Performance decreases when increasing the number of processes due to timestamps verifications.

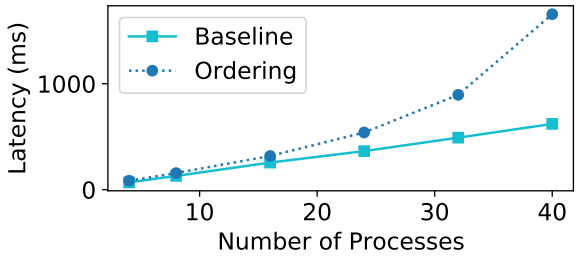

Figure 3.5: Commit latency increases due to the higher number of signed timestamps verified for each proposal.

with an ordering indicator of 0 or 1, and performance stabilises. We measure a value of 350 milliseconds inside the same datacenter, and a value of 450 milliseconds when communications occur between datacenters. We observe a higher χ in the multiple datacenters setup due to higher latencies between distinct regions.

Scalability

We increase the number of processes and measure the throughput and the latency of the blockchain. Throughput is the number of committed transactions per second, whereas latency is the average time required to commit a proposal. Figure 3.4 and Figure 3.5 show that our blockchain does not scale as well as its baseline. This is due to the fact that with an increasing number of processes, each proposal requires an increased number of signed timestamps that must be verified by all processes. For each block, each process verifies a number of proposals proportional to the number of processes, and each proposal contains a number of signed timestamps that are also proportional to the number of processes. The increased cryptographic cost of ordering linearizability is, therefore, quadratic in the number of processes.

3.6 Conclusion

In this chapter, we presented a new ordered reliable broadcast protocol and a fast ordered Byzantine consensus protocol. We demonstrated their practicality by building a blockchain upon them and showing that our blockchain can commit tens of thousands of transactions per second on 40 geo-distributed machines. Interestingly, Pompē [13], which relies on the HotStuff [61] leader-based consensus algorithm, does not scale due to the intrinsic bottleneck introduced by the use of a leader. Despite relying on a leaderless consensus algorithm, the protocol presented in this chapter also fails to scale due to a number of cryptographic verifications that are quadratic with the number of processes. Furthermore, although our protocol almost halves the latency of Pompē, it remains unclear whether the latency could be further reduced. In the next chapter, we introduce Lyra, an SMR protocol that ensures ordering linearizability, is scalable, and has optimal latency.

Chapter 4

Fast and Scalable Resilience to Reordering Attacks

The consensus protocol in Pompē is denoted *Byzantine ordered consensus* (BOC). A BOC protocol is equivalent to a one-shot Byzantine broadcast protocol that outputs a transaction t and an ordering indicator s used to order t. Unfortunately, due to its leader-based design, Pompē experiences a bottleneck that prevents it from scaling [13]. Even with a leaderless design, our experiments in Chapter 3 show that scalability is impeded by a number of signature verifications that is quadratic in the number of processes when all processes submit transactions. More importantly, ordering linearizability alone does not prevent front-running attacks. Kelkar et al. [16] present a front-running attack that can be executed against both the relative and absolute ordering models. The attack is based on violations of the triangle inequality in networks whereby an attacker observes a transaction in the clear and exploits the lack of triangle inequality between network latencies (see Figure 4.1). To prevent reordering attacks, [81] suggests revealing the payload of a transaction only when the transaction can no longer be preempted. This approach is implemented in Fino [17], where the transaction payload is obfuscated and revealed only once the transaction has been committed. However, Fino is a leader-based protocol, and as such, it does not prevent a malicious leader from omitting transactions from up to f processes. Although the underlying DAG transport [116] may resubmit the transaction t later, t has effectively been reordered. Thus, Fino implements *blind order-fairness*.

In this chapter, we present Lyra, a leaderless SMR protocol that prevents the reordering of transactions by combining an order-fair algorithm for BOC with a commit-reveal scheme to obfuscate transactions and only reveal their payloads after they have become part of a stable and immutable set of transactions. Being leaderless, Lyra does not rely on a leader to drive progress. Rather, consensus outputs and SMR progress are decided by each process based on inputs from all processes. Thus, Lyra achieves scalability by leveraging an inherently distributed protocol and a novel approach for determining ordering indicators. As a result, Lyra improves on both the latency and throughput of previous solutions. Because Lyra is leaderless, it is also resilient to censorship by Byzantine leaders. To achieve subsecond commit latency, Lyra introduces a protocol for BOC that executes in 3 message delays in the good

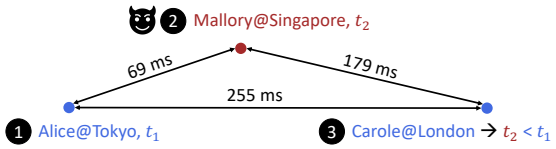

Figure 4.1: Violations of the triangle inequality observed in networks (for latencies measured on AWS regions) allow for reordering attacks despite fair ordering: (1) Alice (A) is in Tokyo and broadcasts a transaction t_1. (2) Mallory (M), in Singapore, receives t_1, observes its content and sends t_2 to Carole to make a profit. (3) Carole (C) receives t_2 before receiving t_1 because $\mathsf{ping}(A, M) + \mathsf{ping}(M, C) < \mathsf{ping}(A, C)$.

case, which we prove optimal for $n > 3f$ (Section 4.3). To achieve a good-case latency of 3 rounds, processes exchange information about network latencies in order to be able to predict the ordering indicators that are perceived by other processes during synchronous periods. Then, to extend Lyra into a leaderless SMR while circumventing the impossibility results arising when committing blocks [12, 96], we introduce a Commit protocol (Section 4.4) to commit the transactions output by instances of BOC. Our Commit protocol is close to approaches adopted in recent DAG-based solutions [116, 117] as it relies on processes piggybacking information and locally deducing the transactions that can be committed. Its main difference resides in the use of consensus rather than reliable broadcast to enable order-fairness and to reduce commit latency.

To summarize, the contributions presented in this chapter are as follows:

1. We propose Lyra, a leaderless BOC protocol for partial synchrony [40] that has optimal good-case latency, optimal Byzantine resilience (i.e. $n > 3f$), and provides transaction obfuscation.

2. We extend Lyra into a leaderless SMR protocol (Section 4.4) that is resilient to harmful reordering attacks and, in particular, to the influence of Byzantine leaders.

3. We implemented Lyra and compared it to Pompē (Section 4.5). Our preliminary results show the potential of our approach in terms of performance and scalability.

Chapter Outline. The rest of the chapter is organized as follows. We describe our computational model in Section 4.1. In Section 4.2, we show the lower bound on the good-case latency to solve the BOC problem. In Section 4.3, we present our Lyra protocol for BOC. We extend Lyra into an SMR in Section 4.4. In Section 4.5, we present our experimental evaluation. We summarize the chapter and conclude in Section 4.6.

4.1 Preliminaries

4.1.1 Network

Recall that in the partially synchronous model [40], messages can be delayed, including by an adversary, until a *global stabilization time* (GST). The value of GST is unknown to correct

processes and can be arbitrarily large. After GST, the network behaves synchronously, and the message delays between any pair of correct processes are bounded by a value Δ. Note that although GST is unknown, Δ is known. The protocols presented in this chapter leverage Δ to create fast paths during synchronous periods while preserving safety during asynchronous periods. Specifically, the value of Δ is used in the Validating Value Broadcast (Section 4.3.1) to ensure progress and in the Commit protocol (Section 4.4.3) to determine the acceptance window.

4.1.2 Byzantine Broadcast

In SMR, all correct processes must agree on a totally ordered set of transactions [29]. One way to implement an SMR is to have processes continuously execute instances of *Byzantine broadcast* where a process broadcasts a transaction, and all correct processes must output the same transaction. We borrow the definition of Byzantine broadcast from [21].

Definition 4.1 (Byzantine Broadcast Problem)**.** *In a Byzantine broadcast protocol, a broadcaster inputs a transaction, and all correct processes must output the same transaction. The transaction output can be empty (i.e., $\perp$). The protocol must ensure the following properties.*

- ***BB-Validity****. If the broadcaster is correct and broadcasts its transaction after GST, then all correct processes output the broadcaster's transaction.*
- ***BB-Agreement****. All correct processes output the same transaction.*
- ***BB-Termination****. All correct processes eventually output a transaction.*

4.1.3 Ordering Clocks

In this chapter, processes use the Clock implementation of the Rank function and thus assign timestamps to the transactions that they receive. Because we use a commit-reveal scheme, processes actually assign a timestamp to the ciphertext c_t of transaction t. Upon receiving c_t, process p_i assigns to t the value $s = \mathsf{Clock}_i(t)$ of its local ordering clock. Formally, for each process $p_i \in P$, we define the function Clock_i that returns the value of p_i's ordering clock when p_i receives c_t:

$$\begin{aligned} \mathsf{Clock}_i \colon \mathcal{T} &\longrightarrow \mathbb{N}, \\ c_t &\longmapsto \mathsf{Clock}_i(t). \end{aligned}$$

4.1.4 Problem

To implement a totally ordered set of transactions, one solution is for processes to agree on a *decided timestamp* for each transaction. Due to varying propagation delays between processes, a transaction t can be perceived with distinct timestamps by distinct processes. Thus, an agreement protocol is required to decide a unique timestamp for t. After that, transactions can be ordered by processes using their decided timestamps.

Definition 4.2 (Decided Timestamp)**.** *The decided timestamp s of a transaction t is a unique timestamp that all correct processes use when ordering t.*

Our main goal is to decide for each transaction a unique timestamp agreed upon by all correct processes so that all the transactions output by the protocol can be ordered.

4.1.5 Lower Bounded Timestamps

In ordering linearizability, each transaction is assigned an ordering indicator that is upper-bounded and lower-bounded by the values of ordering indicators perceived by correct processes. We simplify ordering linearizability and only require that an ordering indicator that is decided be *lower bounded* by the values perceived by correct processes. A lower bound on decided timestamps is necessary to prevent an adversary from affecting transactions that have already been submitted or sequenced. On the other hand, an upper bound only prevents an adversary from delaying the execution of its own transactions and can be achieved anyway if the adversary simply waits before submitting its transactions.

Definition 4.3 (Lower Bounded)**.** *A timestamp s decided for a transaction t is lower bounded with respect to a security parameter $\epsilon > 0$ if and only if $s \geq \mathsf{MinRank}(t) - \epsilon$.*

Because the protocol presented in this chapter relies on predictions of timestamps, our definition includes a margin of error ϵ to account for variations of network delays (see Section 4.3.2).

4.1.6 The BOC Problem

In this section, we define the Byzantine Ordered Consensus (BOC) problem using a reduction from Byzantine broadcast.

Definition 4.4 (BOC Problem)**.** *A BOC protocol is a Byzantine broadcast protocol where a broadcaster inputs a 2-tuple $t' = (t, s)$ that includes a transaction t and a timestamp s, and where all correct processes output either the broadcaster's value t' or the empty value $\perp$. The protocol has the following properties:*

- ***BOC-Validity***. *Conjunction of BB-Validity (cf. Definition 4.1) and if a correct process outputs $t' = (t, s)$, then s is lower bounded.*
- ***BB-Agreement***. *Same as in Definition 4.1.*
- ***BB-Termination***. *Same as in Definition 4.1.*

A BOC protocol is a Byzantine broadcast protocol where a broadcaster broadcasts its input $t' = (t, s)$ consisting of a transaction t and its requested timestamp s, and the transaction and its timestamp are either accepted (i.e., correct processes output t') or rejected (i.e., correct processes output $\perp$). For each execution of the protocol, we denote by $\mathcal{T}_A \subseteq \mathcal{T}$ the set of all accepted transactions, and by $\mathcal{T}_R \subseteq \mathcal{T}$ the set of all rejected transactions, with $\mathcal{T}_A \cap \mathcal{T}_R = \emptyset$. The set of accepted transactions $\mathcal{T}_A$ is a subset of the union of all the transactions that are submitted by processes.

4.2 Optimality in Byzantine Ordered Consensus

In this section, we show that the minimal good-case latency of any solution to the BOC problem is at least 3 communication rounds, which motivates the need for a more efficient protocol than Pompē with 11 rounds [1]. Our definition of good-case latency is borrowed from [21].

Definition 4.5 (Good-Case Latency)**.** *A BOC protocol has a good-case latency of R rounds if all correct processes decide within R rounds after GST when the broadcaster of the transaction is correct.*

Lemma 4.1 (Minimal Latency)**.** *The good-case latency of a BOC protocol that is resilient to $f < \frac{n}{3}$ Byzantine processes is greater than or equal to 3 rounds.*

Proof. For BOC, a broadcaster submits a transaction t and a tentative timestamp s that are either accepted or rejected. If we denote by $t' = (t, s)$ the compound transaction, BOC can be reduced to Byzantine broadcast because it has the same termination and agreement requirements. The good-case latency of Byzantine broadcast in partial synchrony is presented in [21], Theorem 7, and is greater than or equal to 3 rounds when $3f + 1 \leq n \leq 5f - 1$. Therefore, a BOC protocol that is resilient to $f < \frac{n}{3}$ Byzantine processes also has a good-case latency that is greater than or equal to 3 rounds. □

To achieve minimal latency, the broadcaster must include a timestamp with its transaction. Intuitively, if the requested timestamp s is input after the first round, then due to [21], the protocol would require 3 additional rounds to reach agreement on the value of s. In order to request a timestamp that is actually lower-bounded, the broadcaster must predict the timestamps perceived by other processes. The protocol presented in Section 4.3 relies on predicting perceived timestamps and validating these predictions.

4.3 Lyra: Implementation of Ordered Consensus

In this section, we present Lyra[1], a partially synchronous algorithm for BOC. To show that Lyra has optimal good-case latency (Theorem 4.3), we prove that it solves the BOC problem (Theorem 4.2), with a good-case latency of 3 rounds (Lemma 4.3), and that therefore the lower bound in Lemma 4.1 is tight. To implement a protocol with optimal good-case latency, we modify an existing protocol for binary consensus by replacing its broadcast protocol. Our new broadcast protocol, named Validating Value Broadcast, is designed to reliably broadcast the transaction of the broadcaster without introducing additional message delays (Theorem 4.1).

4.3.1 Validating Value Broadcast

The *Validating Value Broadcast* (VVB) protocol is a new broadcast protocol that combines the reliable broadcast [113] of a message with its validation by a Byzantine quorum [118].

[1] In the temple of Apollo at Delphi, observation of the Lyra constellation was used to determine the time to consult the Delphic oracle. The name Lyra stems from the fact that our protocol uses observation of network latencies to predict timestamps.

The protocol is an extension of the *Binary Value Broadcast* protocol [31], a reliable broadcast abstraction for binary values. The Binary Value Broadcast protocol is used in the DBFT binary consensus protocol [33] by processes to broadcast their inputs. To our knowledge, this is the only blockchain consensus protocol that has been fully formally verified [119, 120]. The aim of our new VVB protocol is to add functionalities to the Binary Value Broadcast protocol, but without introducing message delays. In addition to a binary value, our variant can reliably deliver any associated message to all correct processes, while at the same time implementing quorum validation. The modified binary consensus protocol is used to accept or reject the transaction and tentative timestamp of a broadcaster.

Properties

In order for Lyra to solve consensus, the VVB protocol must have the *BV-Termination*, *BV-Uniformity*, *BV-Obligation*, and *BV-Validation* properties present in the definition of Binary Value Broadcast [31]. Additionally, our variant provides a *VVB-Unicity* property that prevents broadcasters from equivocating. Quorum validation is implemented via a configurable function, named ValidatingFunction, that returns either 0 or 1. If the function returns 1 at process p_i for message m, then we say that p_i has *validated* m. The *VVB-Supermajority* property guarantees that if $(1, m)$ is delivered by a correct process, then at least $2f + 1$ processes have validated m.

A process initiates the VVB protocol by broadcasting a message m to all processes and terminates. After that, correct processes may deliver from the VVB protocol either the 2-tuple $(1, m)$, the 2-tuple $(0, \perp)$, or both. Note that the VVB protocol is used as an asynchronous broadcast protocol by the binary consensus protocol and that at any given time, the values delivered by two distinct processes may differ. These divergences are handled by the binary consensus protocol. The protocol is defined by the following properties, where $b \in \{0, 1\}$ and m can be any message.

- **VVB-Termination**. An invocation of the protocol by a correct process always terminates.
- **VVB-Validity**. If a correct process delivers (b, m), then some process broadcast (b, m).
- **VVB-Uniformity**. If (b, m) is delivered by a correct process, then (b, m) is eventually delivered by all correct processes.
- **VVB-Obligation**. Each correct process eventually delivers some values (b, m).
- **VVB-Unicity**. If a correct process delivers (b, m), then no other message $m' \neq m$ can be delivered with b.
- **VVB-Supermajority**. If a correct process delivers $(1, m)$, then at least $2f + 1$ processes validated m.

The first four properties imply the necessary properties of the Binary Value Broadcast for DBFT to solve consensus. Hence, any broadcast protocol satisfying these properties can replace the Binary Value Broadcast in the DBFT consensus protocol.

Implementation

The VV-Broadcast protocol presented in Algorithm 4.1 implements the VVB protocol. First, a process signs its message with its private key (line 2) and broadcasts it in an INIT message (line 3). The message m is signed to prevent equivocation. Upon receiving an INIT message that is correctly signed (line 5), a process p_i tries to validate the message (line 6). If the result of the validation is successful, p_i starts an expiration timer $E = 2\Delta$ (line 7) and broadcasts the value 1 (line 9). Otherwise, p_i broadcasts the value 0 (line 11).

For VVB-Unicity, a correct process only broadcasts the binary value 1 once for each instance of the protocol. Thus, if a message m is delivered with the value 1, then no other message $m' \neq m$ can be validated by more than $2f$ processes and be delivered with the value 1. Additionally, when a correct process broadcasts the binary value 1, it also broadcasts a signature share π_m of the value m that it validated. As a result, when a correct process p_i receives $n - f \geq 2f + 1$ signature shares for the message m, p_i can deliver $(1, m)$ (line 15) and broadcast a proof so that all correct processes eventually deliver $(1, m)$ (line 19).

To implement the VVB-Obligation property in case a message obtains less than the required quorum, a correct process will broadcast the value 0 and the message of the broadcaster after the expiration timeout (line 25). This will cause the value 0 to be eventually delivered to ensure progress.

Theorem 4.1 (Validating Value Broadcast)**.** *The* VV-Broadcast *algorithm (cf. Algorithm 4.1) implements the VVB protocol.*

Proof. We prove each property of the VVB separately:

- **VVB-Termination.** Termination is ensured by the termination of the signing and broadcast functions.

- **VVB-Validity.** When (b, m) is delivered by a correct process p_i, either p_i has received messages from at least $n - f$ processes, or a proof. In both cases, at least $f + 1$ correct processes verified m.

- **VVB-Uniformity.** If $(0, \perp)$ is delivered by a correct process, it was received from at least $n - f$ processes, and thus from at least $f + 1$ correct processes that broadcast 0 to all processes. As a result, all correct processes receive the value 0 from at least $f + 1$ processes and rebroadcast 0, and 0 is eventually delivered by all correct processes. If $(1, m)$ is delivered by a correct process, this process has either received a proof or built the proof itself, and it broadcasts the proof so that all correct processes eventually deliver $(1, m)$.

- **VVB-Obligation.** The protocol is only started by a correctly signed message, and this message is forwarded to all processes after a timeout by any correct process that receives it. Therefore, all correct processes will set an expiration timeout to broadcast 0 if no other value was delivered, and they will eventually deliver 0.

Algorithm 4.1 Validating Value Broadcast

1: **function** VV-Broadcast(m) ▷ *protocol at p_i*
2: $\sigma \leftarrow$ DS.Sign(sk_i, m) ▷ *sign message m*
3: Broadcast(INIT, (m, σ)) ▷ *broadcast signed message*

4: **upon** receiving a message (INIT, (m, σ)) from p_j **do**
5: **if** DS.Verify(pk_j, σ, m) **then** ▷ *verify signature*
6: **if** ValidatingFunction(m) **then** ▷ *message validation*
7: StartTimer(E) ▷ *$E = 2\Delta$*
8: $\pi \leftarrow$ TS.SignShare(tsk_i, m) ▷ *signature share showing p_i has validated m*
9: Broadcast(VOTE, $(1, \pi)$) ▷ *broadcast 1 and signature share*
10: **else**
11: Broadcast(VOTE, 0) ▷ *reject m and broadcast 0 once*

12: **upon** receiving $n - f$ messages (VOTE, $(1, \pi)$) and 1 not delivered **do**
13: $\Pi \leftarrow$ TS.Combine($\{tpk_i\}, \{\pi\}, m$) ▷ *proof of delivery for $(1, m)$*
14: Broadcast(DELIVER, Π) ▷ *broadcast proof to ensure VVB-Uniformity*
15: VVB-Deliver($1, m$) ▷ *deliver $(1, m)$ to VVB*

16: **upon** receiving 1 message (DELIVER, Π) **do**
17: **if** TS.Verify(vk, Π, m) **then**
18: Broadcast(DELIVER, Π) ▷ *rebroadcast proof*
19: VVB-Deliver($1, m$) ▷ *deliver $(1, m)$ to VVB*

20: **upon** receiving $f + 1$ messages (VOTE, 0) **do**
21: Broadcast(VOTE, 0) ▷ *broadcast 0 if not already done*

22: **upon** receiving $n - f$ messages (VOTE, 0) and 0 not delivered **do**
23: VVB-Deliver($0, \perp$) ▷ *deliver $(0, \perp)$ to VVB*

24: **upon** timer expiration $\wedge$ no value delivered **do**
25: Broadcast(VOTE, 0) ▷ *broadcast 0 after expiration of timer*

- **VVB-Unicity.** By design, only the value $\perp$ is delivered with 0. Delivery of $(1, m)$ by a correct process requires the validation of m by at least $2f + 1$ distinct processes, and thus $f + 1$ correct processes. These $f + 1$ correct processes only validate a single value per instance of the protocol, and therefore, any other value $m' \neq m$ obtains at most $2f$ validations.
- **VVB-Supermajority.** This property results directly from the fact that if a correct process delivers $(1, m)$, then it has either received $n - f \geq 2f + 1$ validations for m or a proof that at least $2f + 1$ processes validated m.

□

4.3.2 Prediction and Validation of Timestamps

In this section, we detail how Lyra predicts the timestamps perceived by processes and how processes validate the predictions made by other processes.

Predicting Timestamps

Whenever a broadcaster p_i broadcasts an encrypted transaction c_t, it also stores a reference timestamp $s_{ref} = \mathsf{Clock}_i(t)$ corresponding to the value of p_i's local ordering clock. Then, when a process p_j receives c_t and takes part in the associated consensus instance for t, it piggybacks in its messages to p_i its perceived timestamp $\mathsf{Clock}_j(t)$. This enables p_i to compute the distance $d_{ij} = \mathsf{Clock}_j(t) - s_{ref}$ between p_i and p_j. Note that the distance d_{ij} takes into account the offset between the clocks of p_i and p_j. Each process p_i maintains an array $D_i = \{d_{ij}\}_{j \in [n]}$ of the distances to other processes. Processes can compute the values of the array D_i after a warm-up period where processes broadcast transactions only to measure distances. After that, when a broadcaster wants to broadcast a new transaction t, it computes the set S_t of predicted timestamps and sends S_t along c_t. Values that might be missing from Byzantine processes are filled with a blank value. The timestamp requested for t is the $(n - f)^{th}$ value of S_t.

$$S_t = \{s_{ref} + d_{ij}\}_{j \in [n]}$$

Validation Function

Upon receiving (c_t, S_t), a correct process p_i accepts t and S_t by broadcasting 1 in the associated instance of VVB if and only if the timestamp that was predicted for p_i is correct with respect to the security parameter ϵ.

$$p_i \text{ validates } (c_t, S_t) \Leftrightarrow |\mathsf{Clock}_i(t) - S_t[i]| \leq \epsilon \tag{4.1}$$

Lemma 4.2. *A decided timestamp s that is validated using Equation (4.1) is lower bounded.*

Proof. If s is decided, the VVB-Supermajority ensures that s has been validated by at least $2f + 1$ processes. Thus, S_t contains the correctly predicted timestamps of at least $f + 1$ correct processes. Because s is the $(n - f)^{th}$ value of S_t, there are at most f values in S_t that are greater than s, and s is, therefore, lower bounded by the perceived timestamp of at least one correct process. □

4.3.3 Byzantine Ordered Consensus

Implementation

To implement SMR, Lyra combines, on the one hand, Byzantine agreement to decide whether to output or not each transaction submitted to the protocol, and on the other hand, exchange of information between processes to determine stable prefixes of transactions where not further transaction can be added. Our Lyra algorithm that solves the BOC problem is presented in Algorithm 4.2. First, the broadcaster computes the reference timestamp s_{ref} (line 27) used later to update the distances to other processes, and the set S_t of predicted timestamps for the transaction t (line 29). Then, the broadcaster encrypts t and initiates an instance of binary consensus for t. Once encrypted using threshold encryption, transaction t requires $2f + 1$ decryption shares to be decrypted. Correct processes broadcast their decryption shares for t once t has been committed (cf. Section 4.4.3).

The VV-Propose protocol is presented in Algorithm 4.3. It consists of a modified DBFT [33] protocol for binary consensus where the Binary Value Broadcast has been replaced by the VVB protocol (line 39). The input is $m = (c_t, S_t)$.

Algorithm 4.2 The Lyra Protocol

```
26: function Ordered-Propose(t)                          ▷ protocol at p_i
27:     s_ref ← Clock_i(t)                               ▷ value of p_i's ordering clock
28:     Store(t, s_ref)                                  ▷ used for computing the distances d_ij
29:     S_t ← {s_ref + d_ij}_{j∈[n]}                     ▷ compute clock estimations for t
30:     c_t ← TE.Encrypt(pk, t)                          ▷ obfuscate t
31:     if VV-Propose((c_t, S_t)) = 1 then               ▷ submit c_t and S_t to binary consensus
32:         decide(c_t, S_t)
33:     else
34:         decide ⊥
```

Properties

We now prove that the Lyra protocol ensures liveness during synchronous periods while preserving safety during asynchronous periods.

Theorem 4.2 (Byzantine Ordered Consensus)**.** *The algorithm* Ordered-Propose *(cf. Algorithm 4.2) implements a BOC protocol.*

Proof. We prove each property of Definition 4.4 separately:

- **BB-Termination.** Termination comes directly from the termination property of the underlying binary consensus protocol.
- **BOC-Validity.** The decided timestamp is the $(n - f)^{th}$ value of the set S_t of predicted timestamps. The VVB-Unicity property guarantees that the set S_t is identical at each correct process, and Lemma 4.2 ensures that the decided timestamp is lower-bounded.

Algorithm 4.3 Modified DBFT Protocol

35: **function** VV-Propose(m) ▷ *protocol at p_i*
36: $r \leftarrow 1$
37: $b \leftarrow \perp$
38: **loop**
39: VV-Broadcast$(b, m) \rightarrow$ vvals ▷ *delivered values*
40: StartTimer(Δ)
41: **if** $i = (r \mod n)$ **then** ▷ *p_i is coordinator*
42: **wait until** (vvals $= \{w\}$)
43: Broadcast(COORD$, r, w) \rightarrow c$ ▷ *coordinator broadcast*
44: **wait until** vvals $\neq \varnothing \wedge$ timer expired
45: **if** $c \in$ vvals **then** $e \leftarrow \{c\}$ **else** $e \leftarrow$ vvals ▷ *prioritize coord value*
46: Broadcast(AUX$, r, e) \rightarrow bvals$ ▷ *broadcast these values*
47: **wait until** $\exists s \subseteq bvals$ where the two following conditions hold:
48: • s contains contents received from at least $n - f$ distinct nodes
49: • $\forall v \in s, v \in$ vvals ▷ *every value in s is in* vvals
50: **if** $s = \{v\}$ **then** ▷ *if there is only one value in s*
51: $b \leftarrow v$ ▷ *adopt this singleton value*
52: **if** $v = (r \mod 2)$ and not decided yet **then** decide(v) ▷ *decide*
53: **else** $b \leftarrow (r \mod 2)$ ▷ *otherwise, adopt the current parity bit*
54: **if** decided in round $r - 2$ **then** Exit() ▷ *help others in two last rounds*
55: $r \leftarrow r + 1$ ▷ *increment the round number*
56: **end loop**

- **BB-Agreement.** The agreement property of binary consensus ensures that all correct processes decide the same binary value, and the VVB-Unicity property ensures that if the binary consensus outputs 1, then the same transaction t and the same set of timestamps S_t have been delivered by all correct processes.

□

Latency

In this section, we show that Lyra (cf. Algorithm 4.2) has optimal good-case latency.

Lemma 4.3. *The good-case latency of Lyra (cf. Algorithm 4.2) is 3 rounds.*

Proof. In the good case, the broadcaster is correct, and the network behaves synchronously. Therefore, the broadcaster correctly computes the perceived timestamps of at least $n - f$ correct processes. In the first round, the broadcaster broadcasts the same message (c_t, S_t) to all processes. Then, in the second round, at least $n - f \geq 2f + 1$ correct processes validate c_t and S_t and broadcast 1. Since the network behaves synchronously, correct processes only deliver the value 1 when building a union of $n - f$ responses during the third round (line 47), and therefore decide 1. □

Theorem 4.3 (Optimal Good-Case Latency)**.** *Lyra (cf. Algorithm 4.2) is a BOC protocol with optimal good-case latency.*

Proof. From Lemma 4.1, the good-case latency of BOC is at least 3 rounds when $f < \frac{n}{3}$. Theorem 4.2 shows that Lyra implements a BOC protocol, and Lemma 4.3 shows that Lyra has a good-case latency of 3 rounds. Therefore, Lyra is a BOC protocol with optimal good-case latency. □

4.4 Order-Fair SMR

In this section, we introduce the Commit protocol to extend Lyra into a protocol that solves the SMR problem. Lemma 4.4, Lemma 4.5, and Lemma 4.6 incrementally build stable sets of transactions, and Lemma 4.8 builds upon the committed prefix of Lemma 4.6 and the properties of BOC (Theorem 4.2) to solve the SMR problem. Finally, Lemma 4.7 and Lemma 4.2 extend our SMR with obfuscation and fair-ordering, respectively, to produce the main result of this chapter (Theorem 4.4).

4.4.1 Extending Lyra into an SMR

An SMR protocol requires that all correct processes output messages in the same order. Due to our assumption of partial synchrony (cf. Section 2.2.3), the BOC protocol at process p_i may decide a transaction with a timestamp that is lower than any of the timestamps already decided by p_i. As a result, simply deciding a timestamp for a transaction is not enough for outputting the transaction in SMR. We extend Lyra with the Commit protocol presented in this section

to enable outputting (i.e., committing) transactions. Our solution to an order-fair SMR relies on two protocols:

- the BOC protocol (Section 4.3) decides the set of accepted transactions;
- the Commit protocol (Section 4.4.3) outputs the set of accepted transactions that can be committed.

4.4.2 Definitions

In order to build stable sets of transactions, we formalise the notion of *prefix*. Recall that $\mathcal{T}_A \subseteq \mathcal{T}$ is the set of accepted transactions, and $\mathcal{T}_R \subseteq \mathcal{T}$ is the set of rejected transactions.

Definition 4.6 (Prefix). *The prefix $\Phi(x)$ is the set of accepted transactions whose decided timestamps are less than or equal to x.*

$$\Phi(x) \subseteq \{(t, s) \in \mathcal{T}_A \times \mathbb{N} \mid s \leq x\}$$

To obtain a *stable* prefix, we must guarantee that at some point, no new transaction is added to the prefix. This requires that the prefix eventually becomes *locked* and processes start rejecting new transactions for that prefix.

Definition 4.7 (Locked Prefix). *The prefix $\Phi(x)$ is locked if all* new *transactions whose timestamps are less than or equal to x are rejected. Formally, for any* new *transaction $(t, s) \in \mathcal{T} \times \mathbb{N}$,*

$$(\Phi(x) \text{ is locked}) \wedge (s \leq x) \Rightarrow t \in \mathcal{T}_R.$$

Note that transactions are only accepted after an instance of BOC, and a locked prefix only guarantees that new transactions for the prefix are rejected. This means that pending transactions whose instances of BOC are still running could be added to a locked prefix (i.e., their consensus instances may output 1).

Definition 4.8 (Stable Prefix). *The prefix $\Phi(x)$ is stable if it is locked, and any pending consensus instance for a transaction (t, s), where $s \leq x$, outputs* 0.

Intuitively, a prefix is stable if both new and pending transactions for the prefix are rejected.

Definition 4.9 (Committed Prefix). *A prefix $\Phi(x)$ is committed if it is stable and that it contains all the accepted transactions with a timestamp less than or equal to x.*

$$\Phi(x) \text{ is committed} \Leftrightarrow \Phi(x) = \{(t, s) \in \mathcal{T}_A \times \mathbb{N} \mid s \leq x\}$$

A committed prefix is stable and complete. Hence, it will eventually be committed by all correct processes.

4.4.3 Commit Protocol

The Commit protocol is presented in Algorithm 4.4. The general idea of the protocol is to have processes exchange information about locally locked prefixes and pending transactions in order to deduce globally stable prefixes. A process can then infer committed prefixes using accepted transactions retrieved from a quorum of $2f + 1$ processes. Intuitively, a quorum of size $2f + 1$, combined with the VVB-Supermajority property of VVB, ensures that if a process does not become aware of a transaction, then the transaction will not be accepted.

For locking prefixes, we define the maximum latency $L = 3\Delta$ of BOC as the maximum duration allowed for the execution of an instance of BOC during synchronous periods. Each process has an acceptance window and only accepts transactions whose requested timestamps are not older than L. The validation function is described in line 65. The validating function at p_i checks if the prediction of the clock of p_i is correct (cf. Lemma 4.2) and if the prefix of the timestamp s requested for t is not locally locked (i.e., s resides in p_i's acceptance window). When p_i validates t, p_i also adds t to its set of pending transactions (line 68).

When broadcasting messages, processes piggyback the values of three local variables (line 78):

- $(\mathsf{Clock}_i - L)$: the value of the locally locked prefix (i.e., acceptance window),
- *minPending*: the lowest timestamp validated by the process and whose transaction is still pending,
- A: the local set of accepted transactions (to reduce message size, hash trees can be used for older prefixes).

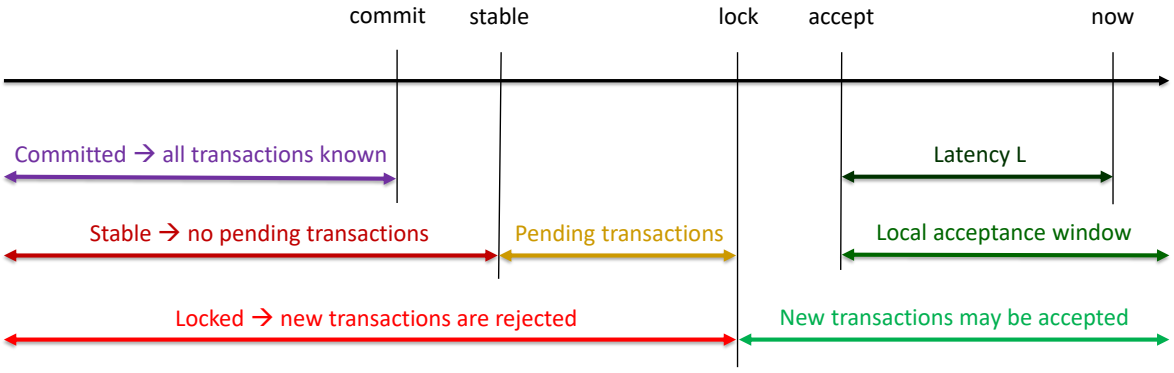

Figure 4.2: Visual representation of the local prefixes and of the acceptance window.

A process that receives these variables (line 80) uses them to compute the prefix $\Phi(\mathit{locked})$ that is locked globally (line 85), the prefix $\Phi(\mathit{stable})$ that is stable (line 87), and the committed prefix $\Phi(\mathit{committed})$ (line 88). Figure 4.2 provides a visual representation of the acceptance window and of the various prefixes.

Note that the value of Δ is only used to determine the local acceptance window of each process and to trigger timeouts, and thus the Commit protocol remains safe even with an

Algorithm 4.4 The Commit Protocol

57: $L \leftarrow 3\Delta$ ▷ *maximum latency*
58: $pending \leftarrow \emptyset$ ▷ *pending transactions*
59: $minPending \leftarrow 0$ ▷ *lowest requested sequence in pending transactions*
60: $A \leftarrow \emptyset$ ▷ *set of accepted transactions*
61: $C \leftarrow \emptyset$ ▷ *transactions to commit*
62: $R \leftarrow \emptyset$ ▷ *values of locally locked prefixes received*
63: $S \leftarrow \emptyset$ ▷ *values of minPending received*
64: $locked \leftarrow 0, stable \leftarrow 0, committed \leftarrow 0$ ▷ *local prefixes*

65: **function** ValidatingFunction(c, S) ▷ *validation function at p_i*
66: $s \leftarrow S[n-f]$ ▷ *requested timestamp is $(n-f)^{th}$ value of S*
67: **if** $|\mathsf{Clock}_i(t) - S[i]| \leq \epsilon \wedge s > (\mathsf{Clock}_i(t) - L)$ **then** ▷ *validation*
68: $pending \leftarrow pending \cup (c, s)$ ▷ *update pending transactions*
69: $minPending \leftarrow \min_{(c,s)\in pending}(s)$ ▷ *update lowest pending transaction*
70: **return** 1
71: **else**
72: **return** 0

73: **upon** BOC output b for transaction (c, s) **do**
74: $pending \leftarrow pending \setminus (c, *)$ ▷ *remove transaction from pending*
75: $minPending \leftarrow \min_{(c,s)\in pending}(s)$ ▷ *update lowest pending transaction*
76: **if** $b = 1$ **then**
77: $A \leftarrow A \cup (c, s)$ ▷ *accept transaction and timestamp*

78: **upon** broadcasting a message m **do**
79: Broadcast$((m, \mathsf{Clock}_i - L, minPending, A))$ ▷ *piggyback local parameters*

80: **upon** receiving a message with $(*, locked_j, min_j, accepted_j)$ from p_j **do** ▷ *handle piggybacked data*
81: $R[j] \leftarrow locked_j$
82: $S[j] \leftarrow min_j$
83: $A \leftarrow A \cup accepted_j$
84: $R_{2f+1} \leftarrow \text{argmax}_{R'\subset R, |R'|=2f+1}\big(\sum_{s\in R'} s\big)$ ▷ *$2f+1$ highest values*
85: $locked \leftarrow \min_{s\in R_{2f+1}}(s)$ ▷ *compute locked prefix*
86: $S_{2f+1} \leftarrow \text{argmax}_{S'\subset S, |S'|=2f+1}\big(\sum_{s\in S'} s\big)$ ▷ *$2f+1$ highest values*
87: $stable \leftarrow \min(locked, \min_{s\in S_{2f+1}}(s))$ ▷ *compute stable prefix*
88: $committed \leftarrow \max_{(c,s)\in A | s\leq stable}(s)$ ▷ *compute committed prefix*
89: TryCommit()

90: **function** TryCommit() ▷ *commit function at p_i*
91: **wait** pending transactions in *committed* are decided **do** ▷ *wait for pending transactions*
92: $committedTxs \leftarrow \{(c, s) \in A \mid s \leq committed \wedge c \notin C\}$ ▷ *committed transactions*
93: $C \leftarrow C \cup committedTxs$ ▷ *commit transactions*
94: **for** c in $committedTxs$ **do**
95: $\rho \leftarrow \mathsf{TE.DecryptShare}(pk, tsk_i, c)$ ▷ *decryption share for c*
96: Broadcast(c, ρ) ▷ *broadcast decryption share for a committed transaction*

asynchronous network. This is because the consensus algorithm is partially synchronous and the locked and stable prefixes are computed using only quorums.

Lemma 4.4. *The prefix* $\Phi(locked)$ *is locked.*

Proof. The prefix $\Phi(locked)$ is the lowest prefix that is locally locked for $2f+1$ distinct processes and, thus, for $f+1$ correct processes. As a result, new transactions for $\Phi(locked)$ can be validated by at most $2f < 2f + 1$ processes and are rejected. □

Lemma 4.5. *The prefix* $\Phi(stable)$ *is stable.*

Proof. By design (line 87), a stable prefix is locked. Thus, we only need to prove that pending transactions for $\Phi(stable)$ are rejected. Any pending transaction t that can be accepted (i.e., $t \in \mathcal{T}_A$) must have been validated by at least $2f + 1$ distinct processes. As a result, t must have been validated by a set Q_i of at least $f + 1$ correct processes that added t to their lists of pending transactions and updated the values of their *minPending* variables accordingly. The prefix $\Phi(stable)$ is computed using the lowest value of *minPending* received from a set of $2f + 1$ distinct processes, and thus from a set Q_j of at least $f + 1$ correct processes. Because we have

$$|Q_i| \geq f + 1 \wedge |Q_j| \geq f + 1 \Rightarrow Q_i \cap Q_j \neq \emptyset,$$

if any transaction t is pending and at the same time $t \in \mathcal{T}_A$, then there is at least one correct process that validated t and whose value of *minPending* is included when computing the variable *stable*. Consequently, any pending transaction whose requested timestamp is not included when computing the value of *stable* is rejected. □

Byzantine processes could prevent prefixes from becoming locked (resp. stable) by sending superficially low values of their locally locked prefixes (resp. lowest pending transactions); to evade this behaviour, in order to compute the variable *locked* (resp. *stable*), processes use the highest $2f+1$ values of locally locked prefixes (resp. lowest pending transactions) received from processes (lines 84 and 86).

Lemma 4.6. *The prefix* $\Phi(committed)$ *is committed.*

Proof. By design (line 88), $\Phi(committed)$ is stable. The set A of accepted transactions at p_i is computed using the values of accepted transactions received from at least $2f + 1$ processes (line 83). As a result, when building the variables *stable* and A, p_i necessarily becomes aware of any transaction that could become part of $\Phi(committed)$ as either pending or as accepted. Due to the BOC-Termination property, p_i only needs to wait for the results of pending transactions in $\Phi(committed)$ (line 91) before committing $\Phi(committed)$. □

Lemma 4.7. *A correct process can decrypt all committed transactions.*

Proof. Each process broadcasts a decryption share (line 96) for all the transactions that it commits. The agreement property of the BOC protocol ensures that all correct processes accept and thus commit the same sets of transactions. Consequently, all correct processes broadcast a decryption share for each committed transaction, and thus, each correct process receives at least $n - f \geq 2f + 1$ valid decryption shares for each committed transaction. □

4.4.4 Order-Fair SMR

In this section, we show that Lyra implements an SMR protocol that ensures order-fairness and transaction obfuscation.

Lemma 4.8 (State Machine Replication)**.** *A protocol where processes continuously input transactions to the Lyra protocol to accept or reject transactions and use the Commit protocol to output committed transactions implements SMR.*

Proof. We prove each property of Definition 2.1 separately:

- **SMR-Safety.** The output consists of the set of committed transactions. By Lemma 4.6, $\Phi(committed)$ contains all the accepted transactions whose timestamps are less than or equal to *committed*, and the agreement property of BOC ensures that all transactions are output with the same timestamps. Therefore, each correct process outputs the same ordered set of transactions.

- **SMR-Liveness.** Because correct processes continuously input their transactions to the protocol, eventually, after GST, correct processes will have their transactions accepted and committed.

□

Theorem 4.4 (Main Result)**.** *Lyra implements an SMR protocol with transaction obfuscation and fair ordering.*

Proof. From Lemma 4.8, Lyra solves the SMR problem. Transaction obfuscation is ensured by Lemma 4.7 and the fact that based on our security assumption (Section 4.1), no process can decrypt a transaction t before t is committed. Theorem 4.2 guarantees a fair ordering of transactions by ensuring that the timestamps of all committed transactions are lower bounded. □

4.4.5 Resilience to Reordering Attacks

A reordering attack against a transaction t_1 is a *harmful reordering attack* if the execution of t_1 is delayed with respect to an execution without any attack, or if a transaction t_2 issued after t_1 and whose content causally depends on t_1 (e.g., front-running) is sequenced before t_1. Note that contrary to front-running or sandwich attacks, we do not classify back-running as a harmful reordering attack because a transaction t_1 can always be trivially back-run by a transaction t_2 if no other transaction is submitted other than t_2. As a leaderless and order-fair protocol, Lyra has no Byzantine leader who can omit some of the transactions or enforce an arbitrary order. This ensures that transactions are not delayed unfairly. Furthermore, the commit-reveal scheme prevents attackers from gaining any knowledge of the payloads of transactions before transactions are committed. As a result, Lyra is a protocol that is resilient to harmful reordering attacks.

4.5 Experimental Evaluation

In this section, we describe our implementation of Lyra and present the results of our experimental evaluation.

4.5.1 Implementation

We implemented a prototype for Lyra using the Rust programming language. The codebase for each node is a little less than 3500 lines of code. In order to compare with Pompē [13], we adopt the same implementation methodologies and use closed-loop clients. Each transaction consists of a unique 32-byte value. Similar to prior work [13, 61, 74], we use batching to amortise the costs of consensus instances. To obfuscate transactions, processes rely on a hash-based commitment scheme [121]. During the benchmark, committed transactions are written in a key-value store.

All of our tests are run on AWS over 3 continents with up to 100 machines. For each data point, we use the same number of servers as in the corresponding Pompē benchmark and virtual machines that are equivalent (Intel Xeon processors, 16 vCPUs, and 64 GB of RAM). We also use the same network distribution where servers are equally distributed between 3 datacenters (Oregon, Ireland, Sydney). We measure both the latency and the throughput of the system. Each client measures the latency of each committed transaction, and we consolidate all these measures to compute the average latency for committed transactions and the throughput of the system.

4.5.2 Benchmark Parameters

We use a batch size of 800 in all of our experiments because this value offers the highest throughput without diminishing the quality of service for clients. A process will instantiate a new instance of BOC either when it has received a full batch of transactions or when a certain amount of time has elapsed since its last proposal. Our experiments over 3 geo-distributed datacenters show that the value of the security parameter ϵ can be reduced to 5 ms without affecting the performance of the system. This means that when requesting a timestamp, a Byzantine process can only deviate from the values perceived by correct processes by up to 5 ms. To allow the computation of the d_{ij} distances, processes piggyback the values of their local clocks when they vote in the VVB protocol. In [122], it is shown that network delays are predictable and that they can be modelled using martingales [123]. A martingale is a discrete-time stochastic process where the expected value of the next observation is equal to the last observation. This means that the most likely value of the network latency d_{ij} between p_i and p_j is the most recent network latency measured between p_i and p_j. Thus, in our experiments, the distance d_{ij} used by p_i to predict the clock of a process p_j is the value of d_{ij} measured by p_i the last time p_i sent a transaction to p_j. This value is computed using the clock value piggybacked by p_j when p_j responds to a transaction sent by p_i.

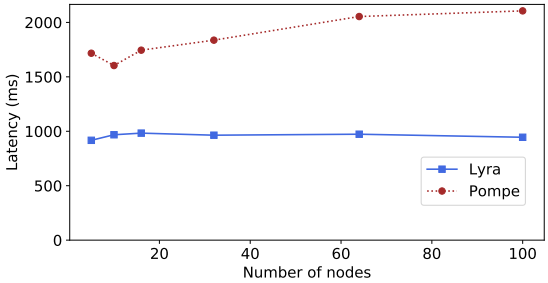

Figure 4.3: Commit Latency. Latency in Lyra is lower than its Pompē baseline due to less message delays.

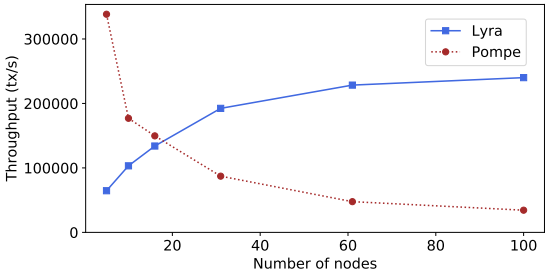

Figure 4.4: Throughput. Pompē performs better up to 20 nodes but does not scale, whereas Lyra scales up to 100 nodes.

4.5.3 Performance Results

Each process runs on a separate node with a dedicated virtual machine. Transactions are submitted by dedicated client nodes apart from the nodes running the consensus protocol. Figure 4.3 and Figure 4.4 show the latency and throughput of the system as a function of the number of nodes ($n \in \{5, 10, 16, 31, 61, 100\}$). The number of nodes for each data point is the same as the number of nodes in the Pompē experiments. The average latency ($< 1s$) is stable when increasing the number of nodes and is half of its baseline Pompē when $n > 60$. While Pompē has a higher throughput when $n \leq 20$, its throughput decreases when increasing the number of nodes. In contrast, the throughput of Lyra increases with the number of nodes and scales up to one hundred geo-distributed nodes. With 100 nodes, Lyra has on average a subsecond commit latency and commits 240,000 transactions per second, thus providing a 2 times improvement for commit latency and a 7 times improvement for throughput compared to Pompē.

The scalability of Lyra comes from the combined effects of broadcasters using tentative timestamps and a leaderless consensus algorithm. The use of signed timestamps in Pompē induces a number of signature verifications that is a quadratic function of the number of processes since each process verifies all the timestamps. In Lyra, only the transaction is signed, and thus, the number of signature verifications is a linear function of the number of processes. This is a linear multiplicative factor improvement over Pompē. Furthermore, the underlying consensus

algorithm in Pompē (HotStuff [61]) is leader-based. As a result, the leader is a bottleneck for the number of transactions that can be processed. In Lyra, all processes can receive transactions and run instances of consensus concurrently.

These preliminary results show the feasibility of our approach and the potential for fast and scalable resilience to reordering attacks in blockchains. We defer to future work for a more comprehensive evaluation of our protocol, including the influence of Byzantine behaviours and a comparison to other protocols.

4.5.4 Byzantine Behaviours

Byzantine processes may try to reduce the chain quality of the output by flooding correct processes with valid transactions. Due to the distributed nature of our protocol, this can be circumvented by having processes allocate network resources fairly between processes. Because processes sign the transactions that they submit, Byzantine processes that submit invalid transactions will be identified eventually. An additional protocol could be implemented to punish these processes.

Using lower-bounded timestamps ensures that Byzantine processes can only drift from the timestamps of correct processes by the value of the security parameter ϵ. Nevertheless, the lack of an upper bound on timestamps enables Byzantine processes to saturate the memory of correct processes by submitting transactions in the future. These transactions must be kept in memory until they are committed. Such behaviour can be mitigated by having processes reject transactions with a timestamp in a too-distant future. Note that using the same algorithms, if we were to assume $f < \frac{n}{4}$ instead of $f < \frac{n}{3}$ and use the median value of S_t instead of the $(n-f)^{th}$ value (Section 4.3.2), then the timestamps decided by Lyra would be both lower bounded and upper bounded and our protocol would then satisfy ordering linearizability.

A network adversary may also delay messages, including insert bias during the warm-up period. But once the propagation delays have become stable, in order to perform a reordering attack, a Byzantine process would need to change the propagation delays d_{ij}, and this unexpected change would trigger the rejection of its transactions.

4.6 Conclusion

In this chapter, we introduced a novel approach that leverages the knowledge of network delays to predict the timestamps that are assigned to transactions by processes. We then built broadcast protocols that integrated this knowledge to achieve a Byzantine ordered consensus protocol that has optimal latency. To implement an SMR, we combined our consensus protocol with a new protocol for determining the sets of transactions that can be committed. Because our protocol combines fair ordering with payload obfuscation and a leaderless design, it prevents harmful reordering attacks. We evaluated our protocol experimentally on a geo-distributed setup and showed that it scales and notably improves the latency and throughput of its baseline Pompē.

Similar to Pompē, Lyra preserves safety during asynchronous periods and ensures liveness and fair ordering during synchronous periods. However, both Pompē and Lyra are vulnerable

to a network adversary. In Pompē, longer message delays can cause the expiration of the timestamps collected by correct processes. In Lyra, unstable networks can cause the requested timestamps to become invalid. In both cases, not only can the adversary cause the rejection of transactions submitted by correct processes, but it can also dictate communication complexity. In Chapter 5, we address these shortcomings by ensuring that all the transactions submitted by correct processes are eventually committed and rely on data dissemination to achieve optimal communication complexity.

Chapter 5

Optimal and Fair Ordering of Financial Transactions

Existing solutions proposed to solve the ordering of transactions either suffer from cyclic ordering dependencies [16, 18] or message delays [13]. Furthermore, they have an $\mathcal{O}(n^3)$ communication complexity [16, 25, 13]. As a result, such solutions can generally not meet the blockchain needs [124]. Relative ordering solutions [12, 16, 18], which can work in asynchronous networks, order transactions by building dependency graphs based on how processes perceive the relative ordering of transactions with respect to each other. Unfortunately, cyclic dependencies often arise when building these graphs of transactions. To avoid this problem, absolute ordering solutions [13, 1, 2] opt instead for assigning a unique timestamp to each transaction. The problem with absolute ordering is that it requires some level of synchrony. They typically discretise time into timeslots and assign transactions to timeslots based on collected timestamps. In an asynchronous network where message delays are unbounded, the risk is that the set of observed timestamps for a transaction expires, hence leading to the violation of the Atomic Broadcast validity property [114]: "if a correct process broadcasts a transaction t, then t is eventually delivered by all correct processes". Furthermore, as depicted in Table 5.1, all solutions proposed for implementing an ordering of transactions [12, 13, 16, 18] either incur an $\mathcal{O}(n^3)$-bit communication complexity per transaction [12, 13, 18], or are designed for synchronous networks [16]. In particular, previous implementations [12, 13, 16, 18] compensate for the absence of a trusted time service by collecting ordering information from $f = \Omega(n)$ processes. Themis [16] has a similar quadratic communication when it only handles 1/4 corruption.[1]

In this chapter, we propose the Asynchronous Ordered Atomic Broadcast (AOAB) protocol that does not suffer from cyclic dependencies or message delays. AOAB is the first asynchronous absolute ordering protocol and the first implementation of order-fairness with optimal communication complexity. Our protocol also satisfies the Atomic Broadcast validity property by ensuring that transactions broadcast by correct processes cannot become obsolete. To achieve

[1] They also show that, in theory, communication may be further reduced by using Succinct Non-Interactive Argument of Knowledge (SNARK) [125]; however, there is still an omitted quadratic term in this calculation, let alone SNARK has its own drawbacks, e.g., heavy proving cost, relying on unfalsifiable computational assumptions, etc.

Table 5.1: Average communication complexity in bits per transaction of existing protocols for order-fairness. ℓ denotes the size of a transaction, and λ is the security parameter.

Protocol	Definition	Corruption	Liveness	P-Sync or Async	Complexity
Aequitas [12]	Block OF*	$n > 4f$	weak	Async	$\mathcal{O}(n^4\ell + \lambda n^4)$
Themis [16]	Deferring OF	$n > 4f$	standard	P-Sync	$\mathcal{O}(n^2\ell + \lambda n^2)^{\ddagger}$
Quick OF [18]	Differential OF	$n > 3f$	weak	Async	$\mathcal{O}(n^2\ell + \lambda n^3)$
Pompē [13]	OL†	$n > 3f$	standard	P-Sync	$\mathcal{O}(n^3\ell + \lambda n^3)$
AOAB	Fair Ordering	$n > 3f$	standard	Async	$\mathcal{O}(n\ell + \lambda n^2)$

*OF: Order Fairness †OL: Ordering Linearizability ‡in the optimistic case, the communication complexity is $\mathcal{O}(n\ell + \lambda n)$

optimal quadratic communication complexity, AOAB takes advantage of threshold signatures to reduce the cost by an extra linear multiplicative factor. In addition, to prevent Byzantine processes from blowing up the cost per transaction, AOAB applies data dissemination [126] to distribute the transactions that are output to all processes. Finally, for Maximal Extractable Value (MEV) resilience, AOAB relies on the encryption-based scheme introduced in [82] and implemented in blockchains in [17] to ensure that the payloads of transactions are only revealed once these transactions have been committed. It is important to note that the AOAB protocol is designed to tolerate up to $f < \frac{n}{3}$ Byzantine failures. This level of fault tolerance aligns with the optimal resilience achievable in an asynchronous network, as determined by the upper bound of resilience proposed by Bracha [113].

In summary, this chapter presents the following contributions:

1. We introduce the first asynchronous implementation of fair ordering that uses absolute ordering indicators. Our protocol preserves broadcast validity as it eventually outputs all the transactions submitted by correct processes.

2. We present the first implementation of order-fairness with optimal communication communication complexity, which makes it resilient to denial-of-service attacks.

3. We show how to integrate payload obfuscation into our protocol. As a result, our protocol is MEV resilient and is suitable for use with financial transactions.

Chapter Outline. The rest of the chapter is organized as follows. In Section 5.1, we provide a formal definition of the AOAB protocol and introduce the broadcast protocols and consensus primitives that are essential to our protocol. Our proposed protocol, AOAB, is presented in Section 5.2, where we not only present the construction of the protocol but also conduct a formal analysis of its security and complexity. In Section 5.3, we show how to make our protocol MEV resilient. We analyze Byzantine behaviours in Section 5.4. We summarise and conclude the chapter in Section 5.5.

5.1 Preliminaries

5.1.1 Asynchronous Network

In this chapter, we assume that the network is asynchronous and that there are no bounds on message delays. This means that the adversary has the ability to arbitrarily delay messages, introducing unpredictable delays in communication. However, it is important to note that messages sent between correct processes will eventually be delivered despite potential delays caused by the adversary.

5.1.2 Atomic Broadcast

In this chapter, to achieve a total order on transactions [22], we rely on the largely studied Atomic Broadcast problem [114]. When a process p_i submits a transaction t to an Atomic Broadcast protocol, we say that p_i *AB-broadcasts* t. When t is output by the protocol, we say that t has been *AB-delivered*.

Definition 5.1 (Atomic Broadcast [114])**.** *An Atomic Broadcast protocol ensures the following properties.*

- ***AB-Validity.*** *If a correct process AB-broadcasts a transaction t, then t is eventually AB-delivered by all correct processes.*
- ***AB-Agreement.*** *If a transaction t is AB-delivered by a correct process, then t is eventually AB-delivered by all correct processes.*
- ***AB-Integrity.*** *A transaction t is AB-delivered by a correct process at most once and only if t was previously AB-broadcast.*
- ***AB-Total-Order.*** *If a transaction t_1 is AB-delivered before a transaction t_2 by a correct process, then all correct processes AB-deliver t_1 before t_2.*

In this chapter, the protocol does not individually determine the delivery of each transaction that is *AB-delivered*. Instead, we divide time into logical epochs, and processes sequentially decide the output of each epoch. In each epoch, correct processes agree on a set of transactions and their associated sequence numbers. This set of transactions extends the output of the SMR. Each process operates within a continuous epoch k and transitions to epoch $k + 1$ when it has completed epoch k. Processes start with epoch 0 and successively decide the transactions that are output in each epoch $e \in \mathbb{N}$. In this chapter, we assume that the epoch number is encoded in a constant number of bits. Note that in practice, this is enough to prevent an overflow because Byzantine processes cannot artificially increment the epoch number (see Section 5.2). Intuitively, each epoch is decided using inputs from a quorum of at least $2f + 1$ processes.

5.1.3 Sequence Numbers

In this chapter, processes use the SeqNum implementation of the Rank function, i.e., Rank = SeqNum. Furthermore, the ordering indicator assigned to a transaction t by a process p_i consists of a pair of elements: the epoch number at p_i combined with the value of p_i's sequence number.

Definition 5.2 (Assigned Sequence Number)**.** *When a process p_i receives a transaction t for the first time, it assigns to t an ordering indicator $o = (e, s)$ where e is the current epoch at p_i and s is the value of p_i's sequence number.*

$$\forall\ p_i \in P,\ \mathsf{SeqNum}_i \colon \mathcal{T} \longrightarrow \mathbb{N} \times \mathbb{N},$$
$$t \longmapsto \mathsf{SeqNum}_i(t) = (e, s).$$

Definition 5.3 (Pair Relation)**.** *We define the* less than *ordering relation, denoted $<$, between two pairs of sequence numbers $o_1 = (e_1, s_1)$ and $o_2 = (e_2, s_2)$ by*

$$o_1 < o_2 \iff (e_1 < e_2) \vee (e_1 = e_2 \wedge s_1 < s_2).$$

For a correct process p_i, at any time, the output of the Atomic Broadcast protocol consists of the transactions that are output by consecutive epochs until the latest known epoch. Inside each epoch, transactions are ordered using the following relation:

Definition 5.4 (Partial Order)**.** *In each epoch, a transaction t_1, with an ordering indicator $o_1 = (e_1, s_1)$, must be executed before a transaction t_2, with an ordering indicator $o_2 = (e_2, s_2)$, if $o_1 < o_2$. We say that t_1 is ordered before t_2 and denote it $t_1 \prec t_2$.*

In our protocol for Asynchronous Ordered Atomic Broadcast (AOAB) (Section 5.2), epochs are decided based on the inputs of processes. For each epoch e, our AOAB protocol comprises two steps: an *ordering step* (Algorithm 5.1) and a *consensus step* (Algorithm 5.2). In the ordering step, a process p_i obtains a sequence number (e, s) for its transaction t and then tries to order t so that t is output in epoch e. During the consensus step, processes decide the transactions that are output in epoch e based on the transactions ordered in e during the ordering step. For the consensus step, each process p_i includes a transaction t with sequence number (e, s) in its submission for epoch e only if p_i has not yet submitted in epoch e (line 41). If p_i has already submitted in epoch e, p_i will include t in its next epoch submission (i.e., in epoch *nextSub*).

Definition 5.5 (Local Process Ordering)**.** *A process p_i orders a transaction t with sequence number (e, s) in epoch $e' = \max(nextSub, e)$ if p_i includes t in its submission for epoch e'. This is denoted $t \sqsubset_i e'$.*

Definition 5.6 (Successful Ordering)**.** *A transaction t is successfully ordered in epoch e if t is ordered in epoch e by at least $2f + 1$ processes, and we denote $t \sqsubset e$.*

$$t \sqsubset e \iff |\{p_i \in P \mid t \sqsubset_i e\}| \geq 2f + 1$$

We can now formalize our definition of a fair ordering.

Definition 5.7 (Fair Ordering)**.** *Let $o_1^{max} = (e_1^{max}, s_1^{max})$ (resp. $o_2^{min} = (e_2^{min}, s_2^{min})$) be the highest (resp. lowest) sequence number assigned by correct processes to a transaction t_1 (resp. t_2). The ordering of the transactions that are output by an Atomic Broadcast protocol is fair if, when o_1^{max} is less than o_2^{min} and t_1 and t_2 are successfully ordered in epochs e_1^{max} and e_2^{min}, respectively, then t_1 is ordered before t_2. Formally, the output of an SMR ensures fair ordering if and only if*

$$o_1^{max} < o_2^{min} \wedge t_1 \sqsubset e_1^{max} \wedge t_2 \sqsubset e_2^{min} \Rightarrow t_1 \prec t_2.$$

In Definition 5.7, the requirement for t_1 and t_2 to be successfully ordered in their assigned epochs means that if a transaction t is not successfully ordered in its assigned epoch e, then it may not be included in the consensus for epoch e and output with the transactions decided in epoch e. However, the BA-validity property ensures that if t is AB-broadcast by a correct process, then t will eventually be output (i.e., in a subsequent epoch).

5.1.4 Broadcast and Consensus Primitives

Multi-valued Validated Byzantine Agreement (MVBA)

The MVBA protocol, as introduced in [127, 25, 100], goes beyond the restriction of binary values and enables agreement on arbitrary values. In this protocol, a public global predicate Q is pre-established and is common knowledge among all participating processes. The form of the predicate function Q is determined based on the specific requirements of the application and can be computed efficiently in polynomial time. The predicate Q is equivalent to an external validity property [25] and is used to determine the validity of the values proposed during the execution of the protocol.

The fundamental concept of the MVBA protocol involves each participating process proposing a value that could incorporate validation information as input. The protocol guarantees that the output value is proposed by at least one process. Hence, the output value also satisfies the predicate Q. All correct processes only input values v to MVBA such that $Q(v) = 1$. Formally, an MVBA protocol satisfies the following properties with all but negligible probability.

- **MVBA-Termination.** If every correct process p_i inputs a value v_i, then every correct process outputs a value.
- **MVBA-External-Validity.** If a correct process outputs a value v, then $Q(v)$ holds.
- **MVBA-Agreement.** For any two distinct correct processes p_i and p_j, if p_i outputs v_i and p_j outputs v_j, respectively, then $v_i = v_j$.

In this chapter, we adopt the MVBA protocol proposed by Lu et al. in [100]. In this MVBA protocol, time complexity is $\mathcal{O}(1)$ and message complexity is $\mathcal{O}(n^2)$. Furthermore, the communication complexity is $\mathcal{O}(n\ell + \lambda n^2)$, where ℓ is the size of the input value and λ is the security parameter.

Provable Reliable Broadcast (PRBC)

As presented in [101], a PRBC protocol extends Bracha's reliable broadcast protocol [113]. We denote PRBC$[id]$ the PRBC protocol instance with identifier id. A correct process submits a value v to the instance id of the protocol using PRBC$[id](v)$. Then, PRBC ensures that each correct process p_i delivers the same value v accompanied by a proof σ that shows that all correct processes will eventually deliver the value v. The delivered proof σ cannot be forged by a polynomially-bounded adversary. We say that p_i PRBC-delivers (v, σ) from PRBC$[id]$. Formally, a PRBC$[id]$ protocol satisfies the following properties except with negligible probability.

- **PRBC-Agreement**. If any two correct processes p_i and p_j PRBC-deliver v and v' from PRBC$[id]$, respectively, then $v = v'$.
- **PRBC-Totality**. If any process PRBC-delivers a valid proof σ from PRBC$[id]$, then every correct process eventually PRBC-delivers v and a valid proof σ from PRBC$[id]$.
- **PRBC-Validity**. If the sender is correct and inputs a value v to PRBC$[id]$, then every correct process PRBC-delivers v and a valid proof σ from PRBC$[id]$.
- **PRBC-Succinctness**. The size of a valid proof σ is independent of the size of the input value v.

In this chapter, we employ the PRBC protocol proposed by Guo et al. in [101]. This specific PRBC protocol exhibits a message complexity of $\mathcal{O}(n^2)$. Furthermore, the communication complexity of the protocol is $\mathcal{O}(n\ell + \lambda n^2)$, where ℓ represents the size of the input value and λ is the security parameter.

Asynchronous Data Dissemination (ADD)

Presented in [126], the ADD protocol ensures that if at least $f + 1$ correct processes receive the same value v as input, while other correct processes start with an initial value $\perp$, then eventually all correct processes output the same value v. This condition holds when there are up to $f = \lceil \frac{n}{3} \rceil - 1$ Byzantine processes. Formally, an ADD protocol satisfies the properties listed below.

- **ADD-Agreement.** For any two distinct correct processes p_i and p_j, if p_i outputs v_i and p_j outputs v_j, respectively, then $v_i = v_j$.
- **ADD-Totality.** If any correct process outputs v, then every correct process outputs v.
- **ADD-Validity.** If at least $f + 1$ correct processes input the same value v and the rest of the correct processes input $\perp$, then every correct process outputs v.

Throughout this chapter, we utilize the ADD protocol proposed by Das et al. in [126]. It has a message complexity of $\mathcal{O}(n^2)$. Additionally, the communication complexity of the protocol is $\mathcal{O}(n\ell + \lambda n^2)$, where ℓ represents the size of the input value and λ is the security parameter.

5.2 Asynchronous Ordered Atomic Broadcast

In this section, we present our Asynchronous Ordered Atomic Broadcast (AOAB) protocol and show that it implements an Atomic Broadcast [114] with a fair ordering of transactions.

High-level overview. Our protocol is designed with two distinct phases for each epoch: the transaction ordering phase and the consensus phase. The transaction ordering phase's objective is to determine the sequence number of each transaction and schedule the transaction so that it is output in its assigned epoch. The sequence number used to order each transaction t is

the median value of a set of $2f + 1$ sequence numbers that have been assigned to t by distinct processes.

The purpose of the consensus phase is to ensure that all correct processes agree on which transactions to output. To achieve optimal communication complexity, we use the hash value of transactions as the input for the consensus phase instead of inputting the transactions themselves. However, to ensure that all the values corresponding to the hashes can be retrieved by all correct processes without causing a significant increase in communication complexity, we employ a strategy during the ordering phase. Specifically, we ensure that for each hash value, its corresponding value has been received by a sufficient number of correct processes during the ordering phase. This guarantees that even if some correct processes did not receive this value during the ordering phase, these correct processes can still receive the value by the end of the consensus phase without blowing up communication complexity.

5.2.1 Implementation

In this section, we describe the construction of our protocol. During each consecutive epoch e, starting with $e = 0$, the protocol decides a set of transactions extending the SMR output. Each epoch e comprises an ordering phase (cf. Algorithm 5.1) and an agreement phase (cf. Algorithm 5.2). In particular, the consensus phase yields a collection of transactions in each epoch, each accompanied by its respective sequence number. Subsequently, these delivered transactions are output (AOAB-delivered) based on their corresponding sequence numbers. If two transactions t_1 and t_2 are output with sequence numbers $\bar{s}_1$ and $\bar{s}_2$, respectively, and that $\bar{s}_1 = \bar{s}_2$, then t_1 and t_2 are sorted deterministically using a lexicographical order.

The transaction ordering phase consists of four logical phases, and the protocol follows the steps outlined below:

1. *Broadcast transaction.* When a process p_i receives a transaction t from a client, p_i submits t to the AOAB protocol via the AOAB-broadcast method (line 7), this step lets process p_i request a set of sequence numbers for its transaction t (line 8).

2. *Assign sequence number.* Upon receiving such request (line 9) for the first time, processes assign to t a pair (e, s) consisting of the combined values of their epochs and sequence numbers and return this value signed with their private keys. When p_i has collected a set $S[t]$ of $2f + 1$ sequences numbers (line 18), p_i will broadcast $S[t]$ in order to collect threshold signature shares for the median value of $S[t]$.

3. *Decide median.* When processes receive $S[t]$ (line 20), they send back to p_i a threshold signature share of the median value of $S[t]$. When p_i has collected at least $f + 1$ shares for the median value $\bar{s}$ of $S[t]$ (line 28), p_i combines these shares into a full proof Σ and broadcast $(\text{ORDER-REQUEST}, h, \bar{s}, \Sigma)$ to all processes. This is the key step that makes our protocol optimal in terms of communication complexity: a valid full proof Σ can prove that at least $f + 1$ correct processes have received a transaction t that satisfies $\mathsf{h}(t) = h$, and as a result, with the help of ADD (line 55, in Algorithm 5.2), all correct processes can also receive the transaction t.

4. *Add ordered transaction to submission buffer.* Whenever a correct process p_i receives a valid message $(\text{ORDER-REQUEST}, h, \overline{s}, \Sigma)$, p_i adds it to its submission buffer (line 33). The submission buffer of each process contains tokens, where each token represents a transaction that can be reliably delivered by correct processes (via ADD) and that is associated with a sequence number that is verifiably the median value of a set of $2f + 1$ signed sequence numbers. Intuitively, the submission buffers of processes constitute their inputs to the epoch consensus phase.

The aforementioned four phases are detailed in Algorithm 5.1 and constitute the transaction ordering phase of the AOAB protocol. The ordering phase is followed by a consensus phase described in Algorithm 5.2. The consensus phase consists of the three following steps:

1. *Broadcast submission buffer.* During each epoch, processes only submit their submission buffers once. When the size of its submission buffer M reaches a size of $K = \Omega(n)$ (line 40), process p_i submits M to the PRBC protocol.

2. *Invoke MVBA.* Upon receiving a string (j, σ_j) from $\mathsf{PRBC}[j, e]$, processes gather them in their proposal $P[e]$ for epoch e (line 44). When a process p_i has gathered a set $P[e]$ of at least $n - f$ strings for epoch e, p_i inputs its proposal $P[e]$ to the MVBA instance of epoch e. When a valid proposal $P_{decided}$ is selected by MVBA for epoch e, process p_i waits for PRBC-delivery of all related submission buffers $\{(\text{EPOCH-SUBMISSION}, M_k)\}$ and collects them in *decidedBuffers* (line 49); additionally, these corresponding hash values will be included in *decidedHash* (line 57).

3. *Deliver transaction.* For each element $(h, \overline{s}, \Sigma)$ in *decidedBuffers*, if process p_i has received a transaction t that satisfies $\mathsf{h}(t) = h$, then t is input to the ADD function. Otherwise, $\perp$ (null value) is input to ADD. Once the corresponding transaction t is received, it is added to *decidedTxs* as $(t, \overline{s})$. Afterwards, all elements in *decidedTxs* are sorted in ascending order based on their respective sequence numbers $\overline{s}$.

   Finally, when their transactions have been AOAB-delivered (line 60), clients can be notified that their transactions have been committed.

5.2.2 Security analysis

In this section, we present detailed proof of our construction. First, we prove that Algorithm 5.1 and Algorithm 5.2 satisfy all the properties of Atomic Broadcast [114].

Lemma 5.1 (Delivery Guarantee)**.** *For any valid element $(h, \overline{s}, \Sigma)$ in the submission buffer, the protocol guarantees that all correct processes can retrieve the corresponding transaction t such that $\mathsf{h}(t) = h$.*

Proof. A valid $(h, \overline{s}, \Sigma)$ indicates that at least one correct process has received the corresponding $(\text{MEDIAN-REQUEST}, h, S[t])$ message. Moreover, it implies that at least $f + 1$ correct processes have sent $(\text{SEQUENCE-RESPONSE}, h, s, \sigma)$ messages. Therefore, at least $f + 1$ correct processes have received a transaction t satisfying $\mathsf{h}(t) = h$ according to the code (lines 9-16). With the

assistance of the ADD protocol implemented in the code (lines 54-58), all correct processes retrieve the corresponding transaction t. □

Lemma 5.2 (Inclusion Guarantee)**.** *If a transaction t is successfully ordered in epoch e, then t is* AOAB*-delivered in epoch e.*

Proof. If t is ordered in epoch e by at least $2f + 1$ processes, then at least $f + 1$ correct processes include t in their submission buffers for epoch e. Because of the predicate Q (line 39), the proposal $P_{decided}$ output by MVBA for epoch e contains valid submission buffers from at least $2f + 1$ distinct processes, out of which at least $f + 1$ are correct. Since two sets of $f + 1$ correct processes necessarily intersect, the decided proposal of epoch e includes at least one correct process that included t in its submission buffer. □

Theorem 5.1. *The* AOAB *protocol (Algorithm 5.1 and Algorithm 5.2) implements an Atomic Broadcast protocol (cf. Definition 5.1).*

Proof. We prove each property separately.

- **AB-Validity.** If a correct process p_i AOAB-broadcasts a transaction t, as $2f+1 \leq n-f$, a correct process p_i will eventually receive a set $S[t]$ of $2f+1$ sequences numbers that are correctly signed for its transaction t. A correct process p_i then eventually collects $f + 1$ shares for the median value $\overline{s}$ of $S[t]$, combines them into a proof Σ, and broadcasts t and Σ. The communication channel ensures that $(t, \overline{s})$ is eventually added to the submission buffers of all correct processes, that is, at least $n - f \geq 2f + 1$ correct processes will add $(t, \overline{s})$ to their submission buffers. Hence, $(t, \overline{s})$ is successfully ordered at this time, and due to Lemma 5.2, t is AOAB-delivered.

- **AB-Agreement.** If a correct process p_i AOAB-delivers a transaction t in an epoch e, then t was output by a corresponding MBVA instance. Lemma 5.1 ensures that t is retrieved by all correct processes, and the MVBA-Agreement property ensures that t is eventually AOAB-delivered by all correct processes in epoch e.

- **AB-Integrity.** Each AOAB-delivered transaction has been submitted by a unique issuer process and comprises a unique nonce from its issuer. Correct processes thus only AOAB-deliver a transaction at most once. To be AOAB-delivered, a transaction must have collected a set of sequence numbers, and by the code, it must then also have been broadcast by some process p_i.

- **AB-Total-Order.** Each correct process AOAB-delivers decided epoch proposals in the same order using an ascending order based on epoch numbers. The MVBA-Termination property ensures that all correct processes can output a proposal for each epoch, and the MVBA-Agreement property ensures that each correct process AOAB-delivers the same set of transactions for each epoch. The transactions in each epoch are also AOAB-delivered in the same order using an ascending order based on their respective sequence numbers, and they are sorted deterministically using a lexicographical order in case of a tie. As a

result, if a transaction t_1 is AOAB-delivered by a correct process before a transaction t_2, then all other correct processes also AOAB-deliver t_1 before t_2.

□

We now prove that our AOAB protocol implements a fair ordering of transactions.

Lemma 5.3 (Correctly Bounded)**.** *The sequence number output by the* AOAB *protocol for a transaction t is upper-bounded and lower-bounded by the sequence numbers assigned to t by correct processes.*

Proof. The sequence number $\overline{s}$ output by the protocol for a transaction t is determined using the median value of a set $S[t]$ of signed sequence numbers assigned to t by $2f + 1$ distinct processes. As there can only be f Byzantine processes, $\overline{s}$ is necessarily upper-bounded and lower-bounded by the sequence numbers assigned to t by at least one correct process. Furthermore, the threshold of $f + 1$ used when building a threshold signature for $\overline{s}$ ensures that at least one correct process has verified that the signatures in $S[t]$ are correct and that $\overline{s}$ is the median of $S[t]$. □

Theorem 5.2 (Fair Ordering)**.** *The* AOAB *protocol implements a fair ordering of transactions (Definition 5.7).*

Proof. From Lemma 5.2, if t_1 and t_2 are successfully ordered in epochs e_1^{max} and e_2^{min}, respectively, then t_1 and t_2 are AOAB-delivered in epochs e_1^{max} and e_2^{min}, respectively. By Lemma 5.3, the sequences numbers (e_1, s_1) and (e_2, s_2) decided for t_1 and t_2, respectively, are upper bounded and lower bounded by sequences numbers assigned by correct processes, and because $(e_1^{max}, s_1^{max}) < (e_2^{min}, s_2^{min})$, we have $(e_1, s_1) < (e_2, s_2)$ and therefore t_1 is ordered before t_2. □

5.2.3 Complexity Analysis

In this section, we provide a comprehensive breakdown of the costs associated with our construction. Recall that ℓ denotes the input size and that λ denotes the security parameter. Note that in a typical financial application, a transaction would carry a client identifier. As a result, ℓ can be influenced by a logarithmic factor relative to the number of clients. However, we make an abstraction of the application domain and simply use ℓ as the input size. Finally, when sending a set of signed sequence numbers, each of size $\mathcal{O}(\lambda)$, we do not account for the process identifier of size $\log(n)$. This is because the sequence number from p_j can simply be the j^{th} element of the set.

Theorem 5.3 (Optimal Communication Complexity)**.** *The average communication complexity is $\mathcal{O}(n\ell + \lambda n^2)$ bits per transaction, which is optimal when $\ell \geq \lambda n$.*

Proof. Consider a process p_i that submits a transaction t of size ℓ to the AOAB protocol so that t can be output. Based on the pseudocode of Algorithm 5.1, the cost breakdown of the ordering phase can be summarized into the following phases.

1. *Broadcast transaction.* In this phase, the sender p_i broadcasts t to all processes in order to collect sequence numbers for t, and thus costs $n\ell$ bits.

2. *Assign sequence number.* During this phase, each process sends for t a signed sequence number of size λ to p_i. In the presence of up to f corrupted processes that may also send the same SEQUENCE-REQUEST message to all, this phase incurs a communication cost of $\mathcal{O}(\lambda n^2)$ bits.

3. *Build median proof.* In this phase, p_i broadcasts to all processes a MEDIAN-REQUEST message containing a set of $\mathcal{O}(n)$ sequence numbers that p_i has collected for t. This step costs λn^2 bits.

4. *Collect median shares.* Each process sends back to p_i a threshold signature of size λ for the median value of the set broadcast by p_i in the previous phase. This incurs a cost of $\mathcal{O}(\lambda n)$ bits.

5. *Order t.* Finally, p_i orders t by broadcasting to all processes a full signature of size λ for the median value of its set $S[t]$. This step also has a cost of $\mathcal{O}(\lambda n)$.

Consequently, the communication complexity per transaction of the ordering phase is $\mathcal{O}(n\ell + \lambda n^2)$. According to the pseudocode of Algorithm 5.2, the cost breakdown of the consensus phase can be split into the following phases.

1. *Broadcast submission buffer.* This phase involves each process submitting its submission buffer to the PRBC protocol. Since each submission buffer contains $\Omega(n)$ transactions, and each transaction is represented by its hash (size λ), a sequence number (size λ), and a proof (size λ), the size of each submission buffer is $\mathcal{O}(\lambda n)$ bits. As a result, the cost of each PRBC is $\mathcal{O}(\lambda n^2)$, and the total cost for this phase is $\mathcal{O}(\lambda n^3)$ bits.

2. *Invoke MVBA.* The size of the input for MVBA is λn, thus resulting in a total cost of $\mathcal{O}(\lambda n^2)$ for this phase.

3. *Deliver transaction.* The cost of this phase is incurred by the invocation of the ADD protocol. The cost of each ADD invocation is $\mathcal{O}(n\ell + \lambda n^2)$. Therefore, the cost for delivering the $\Omega(n)$ transactions decided during the epoch is $\mathcal{O}(n^2\ell + \lambda n^3)$.

This results in a communication complexity of $\mathcal{O}(n^2\ell + \lambda n^3)$ for $\Omega(n)$ transactions, and thus in a communication complexity of $\mathcal{O}(n\ell + \lambda n^2)$ per transaction during the consensus phase. Summing up, the total communication complexity per transaction for the entire protocol is $\mathcal{O}(n\ell + \lambda n^2)$ bits. □

Batching is a common technique where processes submit a batch of transactions instead of a single transaction. This enables the amortisation of the cost of consensus over the batch. Note that our protocol does not use batching and that the cost of $\mathcal{O}(n^2)$ bits is, therefore, a cost per message of size ℓ. Thus, by modifying our protocol to have each process AOAB-broadcast a batch of transactions instead of a single transaction, the cost per transaction can be further reduced.

5.3 AOAB with MEV Resilience

In this section, we enhance our AOAB protocol with MEV resilience. The fundamental idea revolves around integrating two key principles: blind order-fairness and time order-fairness. To achieve MEV resilience, we leverage the commit-reveal framework presented in Fino [118]. In Fino, a process first generates a symmetric encryption key K that it uses to encrypt its transaction t. Then, two distinct methods are employed simultaneously to distribute K to ensure that K can be revealed once t is committed. The first approach involves distributing K using secret sharing [103], while the second approach employs encrypting K using threshold encryption [112]. Ultimately, the participating processes are provided with a secret share of K, a threshold-encrypted version of K, and a hash value h_K of K. As introduced in [25], this approach also serves as a method to achieve secure causal atomic broadcast by preventing the adversary from learning the payloads of transactions before transactions are committed.

Following the commit phase, the framework reveals K, thus enabling all correct processes to decrypt t. Specifically, in Fino, once a transaction is committed, the optimal route is called the *happy path* and involves attempting to reconstruct K by combining at least $f+1$ secret shares of K. If the reconstructed key K' does not align with the hash value h_K, the framework proceeds to generate K' using threshold encryption. For the sake of simplicity, we focus solely on the pessimistic path, which involves utilizing only threshold encryption, foregoing the inclusion of a happy path similar to the one described in the Fino framework. However, for practical purposes, the same happy path as in Fino could easily be incorporated into our protocol.

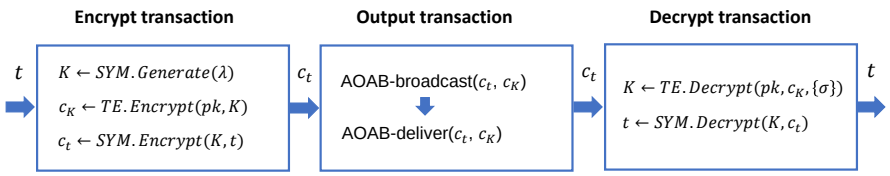

Figure 5.1: Overview of AOAB with MEV Resilience. A process first encrypts its transaction t before submitting it. Once the cipher c_t of t is output, t can be decrypted and committed.

The process of enhancing AOAB with MEV resilience is illustrated in Figure 5.1. Upon receiving a client transaction t, process p_i proceeds as follows.

1. *Encrypt transaction.* Process p_i generates a symmetric encryption key K and uses it to encrypt t, yielding a ciphertext c_t. Process p_i then uses threshold encryption to create an encryption c_K of the key K.

2. *Output transaction.* Process p_i AOAB-broadcasts (c_t, c_K). Whenever a correct process p_j AOAB-delivers (c_t, c_K), p_j generates a threshold decryption share for c_K, and broadcasts it along with the hash of c_t.

3. *Decrypt transaction.* Each correct process first decrypts K using threshold decryption shares for c_K, and then decrypts t using K so that t can be committed.

The MVBA-Termination and MVBA-Agreement properties ensure that all correct processes deliver the same set of transactions in each epoch and, therefore, broadcast a decryption share of c_K, thus ensuring that all correct processes reconstruct K and subsequently decrypt t.

The input size of AOAB is $\mathcal{O}(\ell + \lambda)$. Additionally, the size of both the threshold decryption share and the hash value is $\mathcal{O}(\lambda)$. Consequently, the communication complexity for each transaction is $\mathcal{O}(n(\ell + \lambda) + \lambda n^2) + \mathcal{O}(\lambda) * n * n$, resulting in an overall complexity of $\mathcal{O}(n\ell + \lambda n^2)$. Thus, adding MEV resilience preserves the optimal communication complexity of our protocol.

5.4 Byzantine Behaviours

In Pompē, due to the partially synchronous setting, a transaction broadcast by a correct process can end up being discarded if its collected set expires, for example, if the pre-chosen time bound is shorter than the actual network delay and a network adversary obstructs the ordering of a collected set. By removing the synchrony assumptions, we can ensure the eventual delivery of transactions broadcast by correct processes. This not only improves censorship resilience but also prevents denial-of-service attacks whereby Byzantine processes blow up communication complexity. Furthermore, using a commit-reveal scheme mitigates network attacks as the adversary does not know the payloads of transactions.

Using a sequence number that is the median value of a set of $2f + 1$ sequence numbers ensures that all the sequence numbers output by our protocol are both upper-bounded and lower-bounded by values that have been assigned by correct processes. This approach circumvents the behaviours of Byzantine processes that could assign superficially large or small sequence numbers to the transactions they observe. Note, however, that the ordering guarantee only applies to non-concurrent transactions, i.e., to two transactions t_1 and t_2 such that the sets S_1 and S_2 of sequence numbers assigned by correct processes to t_1 and t_2, respectively, are disjoint. If S_1 and S_2 are not disjoint, we say that t_1 and t_2 are concurrent, and t_1 and t_2 can be output in any order. This results directly from the fact that a transaction t can be output with a sequence number that ranges from the lowest to the highest sequence number assigned to t by a correct process. As a result, although the ordering guarantee of our protocol is conditional only to $f < \frac{n}{3}$, it can be weakened if Byzantine processes induce disparity among the sequence numbers of correct processes. This attack can be mitigated by having correct processes update their sequence numbers by increasing it to a value that is the minimum of a set of $f+1$ observed sequence numbers.

5.5 Conclusion

In this chapter, we leveraged cryptographic, broadcast, and agreement protocols to devise the first asynchronous order-fair protocol with optimal communication complexity and optimal Byzantine resilience. Our protocol improves the communication complexity of previous solutions and implements broadcast validity by ensuring that transactions broadcast by correct processes are eventually committed. We then made our protocol MEV resilient by integrating a commit-reveal scheme that ensures payload obfuscation.

The absolute ordering paradigm introduced by Zhang et al. [13] offers an interesting alternative to the relative ordering paradigm by enabling a more practical approach to ordering. First, circumventing cyclic dependencies enables processes to immediately output transactions without requiring them to wait for potential dependencies. Then, a client only submit its transaction to a single process, whereas in relative ordering a client must broadcast its transaction to all the processes. Thus, although absolute ordering is strictly weaker than relative ordering, it is an appealing trade-off for practical order-fairness. Nevertheless, in Chapter 6, we show that by leveraging trusted hardware, it becomes possible to reduce the theoretical gap between the two ordering paradigms and make the relative ordering method as fair as its absolute counterpart.

Algorithm 5.1 Transaction Ordering Step, code for p_i

1: **State**
2: $epoch \leftarrow 0$ ▷ *consensus epoch*
3: $seq_i \leftarrow 0$ ▷ *local sequence number*
4: $T \leftarrow [\mathcal{T} \rightarrow []]$ ▷ *median threshold shares*
5: $S \leftarrow [\mathcal{T} \rightarrow []]$ ▷ *collected sequence numbers*
6: $M \leftarrow []$ ▷ *submission buffer*

7: **function** AOAB-broadcast t
8: Broadcast(SEQUENCE-REQUEST, t)

9: **upon** receiving (SEQUENCE-REQUEST, t) from p_j **do**
10: **if** t is received for the first time by p_i **then**
11: $s \leftarrow \langle epoch, seq_i \rangle$
12: $seq_i \leftarrow seq_i + 1$
13: $\sigma \leftarrow \mathsf{DS.Sign}(sk_i, \mathsf{h}(t) \mid\mid s)$
14: Broadcast(SEQUENCE-RESPONSE, $\mathsf{h}(t), s, \sigma$)

15: **upon** receiving (SEQUENCE-RESPONSE, h, s, σ) from p_j **do**
16: **if** $\mathsf{DS.Verify}(pk_j, \sigma, \mathsf{h}(t) \mid\mid s)$=1 **then**
17: $S[t] \leftarrow S[t] \cup (j, s, \sigma)$
18: **if** $|S[t]| = 2f + 1$ **then**
19: Broadcast(MEDIAN-REQUEST, $h, S[t]$)

20: **upon** receiving (MEDIAN-REQUEST, $h, S[t]$) from p_j **do**
21: **if** $|S[t]| = 2f + 1 \wedge \mathsf{DS.Verify}(pk_j, \sigma, h \mid\mid s) = 1$ for $\forall\ (j, s, \sigma) \in S[t]$ **then**
22: $\overline{s} \leftarrow \mathsf{Median}(S[t])$
23: $\sigma_{h,\overline{s}} \leftarrow \mathsf{TS.SignShare}(tsk_i, (h, \overline{s}))$
24: send(MEDIAN-RESPONSE, $h, \sigma_{h,\overline{s}}$) to p_j

25: **upon** receiving (MEDIAN-RESPONSE, $h, \overline{s}, \sigma$) from p_j **do**
26: **if** $\mathsf{TS.VerifyShare}(tpk_j, (\mathsf{h}(t), \sigma, \overline{s}))$=1 **then**
27: $T[t] \leftarrow T[t] \cup (j, \sigma)$
28: **if** $|T[t]| = f + 1$ **then**
29: $\Sigma \leftarrow \mathsf{TS.Combine}(\{tpk_i\}, T[t], (h, \overline{s}))$
30: Broadcast(ORDER-REQUEST, $h, \overline{s}, \Sigma$)

31: **upon** receiving (ORDER-REQUEST, $h, \overline{s}, \Sigma$) $\wedge$ $(h, *, *) \notin M$ $\wedge$ $h \notin$ *decidedHash* **do**
32: **if** $\mathsf{TS.Verify}(vk, \Sigma, (h, \overline{s})) = 1$ **then**
33: $M \leftarrow M \cup (h, \overline{s}, \Sigma)$

Algorithm 5.2 Epoch Consensus Step, code for p_i

34: **State**
35: $epoch \leftarrow 0$ ▷ *consensus epoch*
36: $nextSub \leftarrow 0$ ▷ *next submission epoch*
37: $M \leftarrow []$ ▷ *submission buffer*
38: $P \leftarrow []$ ▷ *submissions collected for each epoch*
39: $Q : X \to |X| = n - f \wedge \forall (j,\sigma) \in X, \sigma$ is valid proof for $\mathsf{PRBC}[j,e]$ ▷ *MBVA predicate*
▷ $K = \Omega(n)$
40: **upon** $|M| \geq K \wedge p_i$ has not submitted in *epoch* **do**
41: $\mathsf{PRBC}[i,e](\text{EPOCH-SUBMISSION}, M)$
42: $nextSub \leftarrow nextSub + 1$

43: **upon** PRBC-delivering $(\text{EPOCH-SUBMISSION}, M_j, \sigma_j)$ from $\mathsf{PRBC}[j,e]$ **do**
44: $P[e] \leftarrow P[e] \cup (j, \sigma_j)$
45: **if** $|P[e]| \geq n - f$ **then**
46: $P_{decided} \leftarrow \mathsf{MVBA}[e](P[e])$
47: **for** $\forall\ (k,\sigma) \in P_{decided}$ **do**
48: **wait** until $\mathsf{PRBC}[k,e]$ outputs $(\text{EPOCH-SUBMIS- SION}, M_k)$ **do** ▷ *wait PRBC*
49: $decidedBuffers \leftarrow decidedBuffers \cup M_k$
50: $decidedTxs \leftarrow \emptyset$
51: **for** $(h, \overline{s}, \Sigma) \in decidedBuffers$ **do**
52: **if** received t satisfy $\mathsf{h}(t) = h$ **then**
53: $t \leftarrow \mathsf{ADD}[e](t)$;
54: **else**
55: $t \leftarrow \mathsf{ADD}[e](\perp)$;
56: $decidedTxs \leftarrow decidedTxs \cup (t, \overline{s})$
57: $decidedHash \leftarrow decidedHash \cup (h)$
58: $\mathsf{Sort}(decidedTxs)$
59: **for** $\forall\ t \in decidedTxs$ **do**
60: $\mathsf{AOAB}\text{-DELIVER}(t)$
61: $M \leftarrow M \setminus P_{decided}$ ▷ *remove decided transactions*
62: $decidedBuffers \leftarrow \emptyset$ ▷ *empty buffer*
63: $epoch \leftarrow epoch + 1$ ▷ *increment epoch*

Chapter 6

Resolving the Network Bias in Ordering Linearizability

Multiple solutions [12, 13, 96, 18, 2] have been devised to ensure fairness in the ordering of transactions. The relative ordering paradigm [12, 16, 18] requires clients to submit their transactions to all processes so that transactions can later be ordered using the relative ordering of transactions at a majority of processes. However, having all clients broadcast their transactions is costly and, therefore, impractical. In the absolute ordering paradigm [13], a client only submits its transaction t to a single process p_i so that p_i can forward t to other processes and collect ordering information for t. This new approach is a practical trade-off as it circumvents circular dependencies and broadcasts by clients at the cost of weakened fairness for *isolated* processes. Process isolation is illustrated in Figure 6.1. Due to the network delays between processes, process p_1 is isolated from processes p_2, p_3, p_4: p_2, p_3, p_4 are close to each other and distant from p_1. Consider a scenario where process p_1 broadcasts a transaction t_1 at time $s_1 = 0\,ms$, and where process p_2 broadcasts a transaction t_2 at time $s_2 = s_1 + 50\,ms$. Due to the network distribution of Figure 6.1, a supermajority of $2f + 1 = 3$ processes observe t_2 before t_1 (cf. Chapter 6), and thus t_2 must be ordered before t_1 despite the fact that t_1 was broadcast before t_2. Note that it may sometimes be more interesting to prefer processes with shorter delays in order to improve throughput. In contrast, we focus in this chapter on a solution to determine the sending time of transactions. For some domains, such as decentralized finance, our approach provides more fairness by removing biases due to network delays.

One could think of naively compensating this imbalance by using the knowledge of network delays between processes. A process p_i that receives a transaction t from p_j at a time s_0, and that knows the network delay d_{ji} between p_j and p_i, can compute the time s_t when t was broadcast by p_j using $s_t = s_0 - d_{ji}$. Unfortunately, such a scheme cannot be implemented candidly in the Byzantine model. For instance, a Byzantine process p_B could make its distances to other processes appear larger by delaying the sending of all of its messages by an amount $d_B > 0$. If p_B stops delaying its messages before sending a new transaction t', a process p_i receiving t' at time s'_0 would believe that t' was sent at time $s_{t'} = s'_0 - (d_{ji} + d_B) < s'_0 - d_{ji}$, and t would unfairly preempt earlier transactions. A novelty of our solution is to combine cryptographic

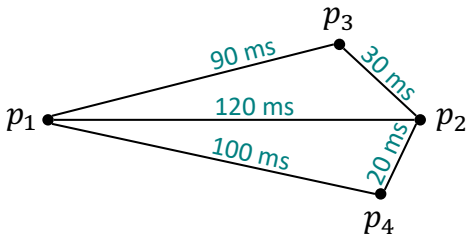

Figure 6.1: Network delays between processes.

	p_1	p_2	p_3	p_4
t_1, $s_1 = 0\,\text{ms}$	$0 + 0 = 0$	$0 + 120 = 120$	$0 + 90 = 90$	$0 + 100 = 100$
t_2, $s_2 = 50\,\text{ms}$	$50 + 120 = 170$	$50 + 0 = 50$	$50 + 30 = 80$	$50 + 20 = 70$

Table 6.1: Times when transactions t_1 and t_2 are received by processes. Process p_1 broadcasts t_1 at time $s_1 = 0\,ms$, whereas p_2 broadcasts t_2 at time $s_2 = 50\,ms$. Reception times are computed by adding the times when transactions are broadcast (i.e., 0 or 50) to the network delays (c.f. Figure 6.1).

challenges [75] with TEEs to determine safe values of network delays.

To prevent process isolation, it would be sufficient to be able to verify network delays. In this chapter, we present Aion[1], a set of leader-based and leaderless protocols that leverage trusted execution environments (TEEs) to solve process isolation. First, we take advantage of the security guarantees provided by TEEs to devise a challenge-response protocol that enables processes to compute safe values of network delays, and we use an additional protocol such as [128] to increase the reliability of TEEs clocks. As far as we know, we propose the first solution that relies on an additional protocol to secure the TEE clocks. The challenges prevent Byzantine processes from advertising network delays that are shorter than the actual network delays. Note that a Byzantine process can still advertise larger network delays simply by retaining messages. Then, we rely on the fact that the network is synchronous most of the time and that network delays are stable [122] and require that measured network delays remain constant. This ensures that if a Byzantine process has inserted biases to increase the values of its network delays, then it has to abide by those new values. As a result, processes can determine the times when transactions are sent in a safe way because it prevents Byzantine processes from preempting older transactions.

Notice that it is not possible to directly use the timestamps provided by a TEE because a Byzantine process could generate transactions with valid timestamps using its TEE and could then broadcast these transactions at a future time in order to preempt transactions that it observes. In the financial domain, this is characterized as a front-running attack [15] and can

[1]Aion is a Hellenistic deity that symbolises a cyclic time. The name Aion stems from the fact that our protocols rely on repeating and constant network delays.

have detrimental economic repercussions. Another novelty of our protocol is that it complements the timestamps provided by TEEs with the verification of network delays to ensure the freshness of transactions. Aion is designed to achieve an accurate ordering of transactions when the network is behaving synchronously. However, networks can suffer various kinds of failure [129] and can behave asynchronously as a result of these failures. In these scenarios, the network delays are unknown, and protocols that assume an upper bound on message delivery may see their safety properties violated [130]. Therefore, we devise partially synchronous protocols in order to preserve safety during asynchronous periods. We discuss the loss of liveness and mitigation strategies during asynchronous periods in Section 6.6.2. Specifically, we make the following contributions:

- We leverage TEEs to devise a protocol that enables processes to safely measure network delays. The novelty of our approach resides in using TEEs so that the measurements of network delays can be used without compromising safety.
- We build upon safe values of network delays and compute the times when transactions are broadcast with proven accuracy during synchronous periods while preserving safety during asynchronous periods.
- We integrate the sending times of transactions with both leader-based and leaderless consensus protocols to implement Aion, a set of SMR protocols with fair ordering. Thus, we show that our approach is modular and practical due to its low overhead of only two message delays.

Chapter Outline. The rest of this chapter is organised as follows. Section 6.1 presents the concepts and the building blocks of our protocol. We introduce the timestamping protocol in Section 6.2 and use it to build the ordering protocol in Section 6.3. We build upon the ordering protocol to build Leader-Based Aion in Section 6.4 and Leaderless Aion in Section 6.5. We discuss our results in Section 6.6, and we conclude in Section 6.7.

6.1 Preliminaries

6.1.1 Network Delays

Recall that in a partially synchronous network, messages can be delayed up to a *global stabilization time* (GST) whose value is unknown. After GST, the network behaves synchronously, and network delays between correct processes are bounded by a known value Δ. During synchronous periods, we assume that the network delays are stable and that the fluctuations in network delays between any two processes are bound by $\epsilon > 0$. This assumption relies on recent studies on the probability distributions of network delays [122]. For stable networks, ϵ is usually less than one millisecond [131]. Let d_{ij} denote the network delay between p_i and p_j. During a synchronous period X, we have

$$\forall s_1, s_2 \in X, \forall p_i, p_j \in P, |(d_{ij} \text{ at } s_1) - (d_{ij} \text{ at } s_2)| \leq \epsilon.$$

6.1.2 Trusted Execution Environments

We assume the existence of Trusted Execution Environment (TEE) technology. A TEE provides a secure environment that ensures that each process executes the protocol correctly. Formally, a TEE guarantees the following properties [132]:

- authenticity of the code executed by each process,
- integrity and confidentiality of runtime states (memory, registers, I/O,...),
- a trusted time service,
- remote attestation in order to prove correctness to a third-party.

We also assume that TEEs are resilient to both software and hardware attacks. As a result, when a process becomes corrupted, the adversary can only take control of resources outside of the TEE (e.g. operating system, network,...). Such technology is implemented, for instance, by hardware with Intel Software Guard Extensions (SGX) [133].

6.1.3 Trusted Clocks

In this section, processes use the Clock implementation of the Rank function. We assume that each process p_i has a trusted local clock denoted $\mathsf{Clock}_i()$ that is managed by the TEE environment and that returns timestamps in $\mathbb{N}$. This is implemented, for instance, in Intel SGX. Although Intel SGX provides access to a secure timer, a privileged user can still manipulate this timer [134]. Consequently, we propose that the local clock be secured with an additional protocol such as TimeSeal [128] to secure the value of $\mathsf{Clock}_i()$. TimeSeal adopts a holistic approach to obtain a reliable time stack by securing the timer, ensuring that the timer can be read in a timely manner, and protecting timekeeping software. During asynchronous periods, the offsets between clocks can grow unboundedly. But after GST, the network is synchronous, and the offsets between any two clocks are bounded by $\delta > 0$. Clock synchronization [135] can achieve a value of δ less than a millisecond, and typically in the order of tens of microseconds [136].

6.1.4 Ordering Intervals

We divide the set $\mathbb{N}$ of all possible timestamps into consecutive intervals of size ℓ. Let I_k denote the k^{th} interval,

$$\forall k \in \mathbb{N}, I_k = [k\ell, (k+1)\ell).$$

We also define the interval mapping function $\mathcal{I}$ that maps any timestamp s to its corresponding interval $\mathcal{I}(s)$ such that $s \in I_{\mathcal{I}(s)}$,

$$\forall s \in \mathbb{N}, \mathcal{I}\colon \mathbb{N} \longrightarrow \mathbb{N},$$
$$s \longmapsto k \mid k\ell \leq s < (k+1)\ell.$$

In the rest of the chapter, when the context is unambiguous, we simply refer to $I_{\mathcal{I}(s)}$ by $\mathcal{I}(s)$. Intervals are used as a basis for implementing a *total order broadcast* [114]: to build a totally ordered set of transactions, correct processes reach agreement on the set of transactions in each interval.

6.1.5 Cryptography

In this section, we assume that the private key of each process resides inside the memory of the TEE and is protected against unauthorized access. Let $\langle v \rangle_i$ denote that the value v has been signed using the private key sk_i of a process p_i.

$$\langle v \rangle_i = \mathsf{DS.Sign}(sk_i, v)$$

6.1.6 Consensus

To decide the content of each interval, we rely on generic consensus abstractions [7]. Such abstractions enable all correct processes to agree on the set of transactions in each interval. We consider protocols that have the classical *termination* and *agreement* properties, but also an *external validity* property [25]. External validity relies on a predicate γ and requires that any decided value contains correctly signed inputs from at least $2f + 1$ distinct processes. More formally, we define the predicate

$$\gamma \colon v \mapsto |v| \geq 2f + 1 \ \wedge \ \forall \langle x \rangle_i \in v, \mathsf{DS.Verify}(pk_i, \langle x \rangle_i) = 1.$$

We assume the following properties for the consensus problem:

- **Termination.** Each correct process eventually decides a value.
- **Agreement.** All correct processes decide the same value.
- **External Validity.** If a correct process decides a value v, then $\gamma(v)$ holds.

For Leader-Based Aion (Section 6.4), we assume the existence of a leader-based consensus protocol, denoted Leader-Propose, where a leader proposes a value, i.e., the sequence of transactions for an interval, and processes agree on whether to output or not the value of the leader. For instance, HotStuff [61] implements such abstraction. For Leaderless Aion (Section 6.5), we assume the existence of a leaderless consensus protocol denoted Leaderless-Propose, where instead of being proposed by a leader, the decided value comes from all processes. Such abstraction is implemented, for instance, by BFT-Archipelago [137].

6.2 Timestamping Protocol

In this section, we present the protocols used by processes to determine the time when a transaction is broadcast. First, the Network Challenge protocol (Section 6.2.1) enables processes to obtain reliable values of network delays. Then, these network delays are used in the Timestamp Validation protocol (Section 6.2.2) to determine if the timestamps requested for transactions are valid.

6.2.1 Network Challenge

The *Network Challenge* protocol is used by processes to measure network delays. To this end, processes regularly challenge other processes by sending them cryptographic nonces. The

Network Challenge protocol is presented in Algorithm 6.1. To prevent Byzantine processes from generating a dictionary of responses to the challenges, and thus to ensure the freshness of the timestamps received, challenges that are sent by a process p_i are signed by p_i (line 6). When a process p_j receives a challenge $\langle u \rangle_i$ from p_i, p_j answers with the signed value $\langle i, u, s \rangle_j$ containing the secure timestamp s generated for u, and the values u and i to certify that the TEE of p_j created the timestamp s for the challenge u sent by p_i (line 12). Upon receiving a response $\langle i, u, s \rangle_j$ to its challenge from p_j, p_i computes the network delay d_{ji} from p_j to p_i using $d_{ji} = \mathsf{Clock}_i() - s$ (line 16). Each process maintains an array D that stores the values of network delays obtained with these challenges.

Algorithm 6.1 Network Challenge Protocol

```
1:  State
2:      U ← []                                          ▷ challenges sent by p_i
3:      D ← []                                          ▷ network delays computed by p_i

4:  function Challenge p_j
5:      u ← Nonce()                                     ▷ generate challenge
6:      ⟨u⟩_i ← DS.Sign(sk_i, u)                        ▷ sign challenge
7:      Send(CHALLENGE, ⟨u⟩_i) to p_j                   ▷ send challenge to p_j
8:      U[j] ← u                                        ▷ store challenge

9:  upon receiving a message (CHALLENGE, ⟨u⟩_j) from p_j do
10:     if DS.Verify(pk_j, ⟨u⟩_j) then                  ▷ verify signature
11:         s ← Clock_i()
12:         ⟨j, u, s⟩_i ← DS.Sign(sk_i, (j, u, s))      ▷ sign response
13:         Send(RESPONSE, ⟨j, u, s⟩_i) to p_j           ▷ respond to p_j's challenge

14: upon receiving a message (RESPONSE, ⟨j, u, s⟩_j) from p_j do
15:     if DS.Verify(pk_j, ⟨j, u, s⟩_j) ∧ C[j] = u then
16:         d_ji = Clock_i() − s                         ▷ compute network delay
17:         D[j] ← d_ji                                 ▷ update network delays
18:         U[j] ←⊥                                     ▷ discard challenge
```

6.2.2 Timestamp Validation

The *Timestamp Validation* protocol enables processes to validate the timestamp of a transaction by inferring the time when the transaction was sent. It combines the network delays obtained via the Network Challenge protocol with a requirement for these message delays to remain constant. The requirement for stable network delays is based on recent studies and experiments on the probability distributions of network delays [122].

Algorithm 6.2 shows the protocol for deciding whether to accept or reject transactions

Algorithm 6.2 Timestamp Validation

```
1: function Validate t, s, j                    ▷ validation of t and s received from p_j
2:     s_send ← Clock_i() − D[j]                ▷ compute send time of t
3:     if |s_send − s| ≤ λ + δ then             ▷ check the timestamp of t
4:         return true                          ▷ accept s for t
5:     else
6:         return false                         ▷ reject s for t
```

based on their requested timestamps. A process simply computes the time when it believes the transaction was sent (line 2) and compares it to the timestamp of the transaction (line 3). During synchronous periods, the error when estimating the send time of a transaction is bounded by $2(\delta + \epsilon)$.

Lemma 6.1. *After GST, a correct process p_i validates a transaction t that has a requested timestamp s (cf. Algorithm 6.2) only if the difference between s and the actual send time s_{actual} of t is less than $2(\epsilon + \delta)$.*

$$\forall p_i \in P^{Corr}, p_i \text{ accepts } (t, s) \Rightarrow |s - s_{actual}| \leq 2(\epsilon + \delta)$$

Proof. If a correct process p_i validates (t, s), then p_i determined that t was sent at a time s_{est}, and that $|s_{est} - s| \leq \epsilon + \delta$. Process p_i computed s_{est} using a challenge, and particularly by using the timestamp included in the response to the challenge and the network delay that p_i computed based on the challenge. After GST, the offsets between the clocks of processes are bounded by δ, and Byzantine processes cannot drift from the expected network latencies by more than a quantity ϵ, so the margin of error for challenges is $\epsilon + \delta$. As a result, the margin of error of s_{est} is $\epsilon + \delta$, and thus $|s - s_{actual}| \leq 2(\epsilon + \delta)$.

□

6.3 Ordering Phase

The *ordering* phase, borrowed from Pompē [13], enables a process p_i to request a timestamp s for a transaction t and to schedule (t, s) so that t is included in $\mathcal{I}(s)$. The aim of the ordering phase is to ensure that if the network is behaving synchronously and that a correct process terminates the ordering phase for (t, s), then t is guaranteed to be included in the interval $\mathcal{I}(s)$. This protocol is used as a preliminary step both in Leader-Based Aion (Section 6.4) and in Leaderless Aion (Section 6.5).

6.3.1 Stability of Committed Intervals

In an SMR, all correct processes output transactions in the same order. Specifically, a transaction is only output when all of its preceding transactions have been delivered. Therefore, after a transaction t is output, no new transaction can be output before t. We refer to this property as the *stability* of committed intervals: once the consensus instances for the k first intervals

have terminated and the sets of transactions in these intervals are known, no other transaction can be added to an interval I_m such that $m \leq k$. Hence, the ordering phase must preserve the stability of committed intervals.

In order to guarantee that the transactions that have been successfully ordered are committed in their corresponding intervals while at the same time preserving the stability of committed intervals, we rely on standard quorum intersections [118]. On the one hand, a transaction (t, s) is successfully ordered in the interval $\mathcal{I}(s)$ if at least $2f + 1$ processes accept to schedule t in $\mathcal{I}(s)$. On the other hand, using the external validity predicate (Section 6.1.6), Aion protocols (Section 6.4, Section 6.5) determine the content of each interval by collecting the transactions that have been ordered by at least $2f+1$ processes. As a result, if a transaction (t, s) is ordered by at least $2f+1$ processes in the interval $\mathcal{I}(s)$, then t is guaranteed to be output in the interval $\mathcal{I}(s)$.

6.3.2 Ordering Protocol

The ordering protocol is presented in Algorithm 6.3. First, a process p_i broadcasts a transaction t and a requested timestamp s (line 7). Processes validate the timestamp s of p_i using the Timestamp Validation protocol (Algorithm 6.2). When a process p_j accepts s, p_j also sends a threshold encryption share $\pi_{t,s}$ (line 10) to p_i. Then, p_i waits until it has collected at least $2f+1$ encryption shares (line 16), and combines these shares into a full proof $\Pi_{t,s}$ (line 17) that it broadcasts. When processes receive the full proof $\Pi_{t,s}$, they verify the aggregated signature (line 20) and reply with the interval where they order (t, s) (lines 23 and 25). The Aion protocols in the following sections (Section 6.4, Section 6.5) determine the content of each interval I_k by collecting the transactions that have been ordered in I_k by at least $2f + 1$ processes. If a process p_j has not yet submitted the transactions that it has ordered for $\mathcal{I}(s)$, then it accepts to order (t, s) in $\mathcal{I}(s)$ (line 23). Otherwise, p_j includes (t, s) in its submission for the next interval (line 25).

Lemma 6.2. *If a transaction (t, s) is ordered in an interval I_k by at least $2f+1$ processes, and if the content of I_k is determined by collecting the transactions ordered in I_k by at least $2f + 1$ processes, then (t, s) is output in I_k.*

Proof. If (t, s) is ordered in I_k by $2f + 1$ processes, then at least $f + 1$ correct processes will include (t, s) in their submissions for I_k. The content of I_k includes submissions from at least $f + 1$ correct processes, and therefore, there is at least one correct process that ordered (t, s) in I_k and whose submission is used for determining the content of I_k. □

Note that if a transaction (t, s) is ordered in $\mathcal{I}(s)$ by less than $2f + 1$ processes, it may or may not be output in $\mathcal{I}(s)$ depending on the set of $2f + 1$ processes whose submissions are used for the interval $\mathcal{I}(s)$. Assume that a transaction does not get ordered in the requested interval by at least $2f + 1$ processes and that the process that requested the ordering receives a set $I = \{I_k\}$ of ordered intervals, where $|I| \geq 2f + 1$. Let I_{min} be the lowest interval among the $f + 1$ highest intervals in I. Then, the transaction is guaranteed to be output no later than in the interval I_{min}.

Algorithm 6.3 Ordering Protocol

State
  $S \leftarrow [\mathcal{C} \rightarrow []]$ ▷ *shares collected by* p_i
  $C \leftarrow [\mathbb{N} \rightarrow []]$ ▷ *transactions ordered for each interval*
  $nextSub \leftarrow 0$ ▷ *next interval submitted by* p_i

function Order (t)
  $s \leftarrow \mathsf{Clock}_i()$
  $\mathsf{Broadcast}(\text{REQUEST}, (t, s))$ ▷ *request timestamp* s *for* t

upon receiving a message $(\text{REQUEST}, (t, s))$ from p_j **do**
  if $\mathsf{Validate}(t, s, j)$ **then**
    $\pi_{t,s} \leftarrow \mathsf{TS.SignShare}(tsk_i, \text{ACCEPT}||t||s)$ ▷ *encryption share of acceptance*
    $\mathsf{Send}(\text{ACCEPT}, t, \pi_{t,s})$ to p_j
  else
    $\mathsf{Send}(\text{REJECT}, t)$ to p_j

upon receiving a message $(\text{ACCEPT}, t, \pi_{t,s})$ from p_j **do**
  $S[t][j] \leftarrow \pi_{t,s}$ ▷ *store share from* p_j
  if $|S[t]| \geq 2f + 1$ **then**
    $\Pi_{t,s} \leftarrow \mathsf{TS.Combine}(\{tsk_j\}, S[t], t)$ ▷ *create proof of acceptance*
    $\mathsf{Broadcast}(\text{ORDER}, t, \Pi_{t,s})$

upon receiving a message $(\text{ORDER}, t, \Pi_{t,s})$ from p_j **do**
  if $\mathsf{TS.Verify}(vk, \Pi_{t,s}, t)$ **then**
    $C[nextSub] \leftarrow C[nextSub] \cup (t, \Pi_{t,s})$
    if $\mathcal{I}(s) \geq I_{nextSub}$ **then**
      $\mathsf{Send}(\text{INTERVAL}, t, \mathcal{I}(s))$ to p_j ▷ p_i *submits* t *in* $\mathcal{I}(s)$
    else
      $\mathsf{Send}(\text{INTERVAL}, t, I_{nextSub})$ to p_j ▷ p_i *submits* t *in* $I_{nextSub}$

6.4 Leader-Based Aion

In this section, we present Leader-Based Aion, an order-fair SMR protocol, by integrating the previous ordering step (Section 6.3) with a leader-based consensus protocol. Our leader-based protocol is analogous to Pompē [13] and consists of (1) an ordering step (Section 6.3) that is executed continuously by processes, and (2) a consensus step for each interval. During synchronous periods, a correct process p_i successfully orders a transaction (t, s), and all correct processes have received a full proof $\Pi_{t,s}$ and ordered (t, s) in the interval $\mathcal{I}(s)$ after 3 rounds (cf. Algorithm 6.3). Consequently, processes can start the agreement protocol to decide the content of an interval $I_k = [k\ell, (k+1)\ell)$ when their clocks reach the value $(k+1)\ell + 3\Delta$.

Algorithm 6.4 Leader-Based Aion

```
1:  State
2:      C ← [ℕ → []]                          ▷ transactions ordered for each interval
3:      L ← [ℕ → []]                          ▷ submissions collected for each interval
4:      nextSub ← 0                           ▷ next interval submitted by p_i
5:      collecting ← false

6:  upon Clock_i() ≥ (k+1)ℓ + 3Δ do           ▷ I_k can be decided
7:      if i ≡ k (mod n) then                 ▷ p_i is the leader of I_k
8:          collecting ← true
9:          Broadcast(COLLECT, k)

10: upon receiving a message (COLLECT, k) from p_j do
11:     if j ≡ k (mod n) then                 ▷ verify leader
12:         wait Clock_i() ≥ (k+1)ℓ + 3Δ do   ▷ wait for interval
13:         ⟨C_k⟩ ← DS.Sign(sk_i, C[k])       ▷ sign submission
14:         Send(SUBMIT, ⟨C_k⟩) to p_j         ▷ send submission to leader
15:         nextSub ← nextSub + 1

16: upon receiving a message (SUBMIT, C_k) from p_j do
17:     if collecting ∧ DS.Verify(pk_j, C_k) then   ▷ verify signature
18:         L[k] ← L[k] ∪ C_k                 ▷ add submission to proposal
19:         if |L[k]| ≥ 2f + 1 then
20:             collecting ← false
21:             Leader-Propose(k, L[k])       ▷ leader proposes L[k] for I_k
```

The Leader-Based Aion protocol is presented in Algorithm 6.4. When a process p_i learns that the agreement protocol for the interval I_k can be started (line 6), and that p_i is the leader for the interval I_k, p_i broadcasts a COLLECT message to start collecting submissions for I_k (line 9). In response, processes sign their submissions (line 13) before sending them to p_i. The signatures are used to verify that the proposal of p_i for I_k contains submissions from at least

$2f + 1$ distinct processes and that, therefore, it satisfies external validity (cf. Section 6.1.6). When p_i has collected at least $2f + 1$ submissions (line 19), p_i starts a consensus instance over its proposal for I_k (line 21). If the leader is Byzantine, processes can deterministically decide on a new leader in case the proposal is invalid or the absence thereof.

Theorem 6.1. *A protocol using Algorithm 6.4 to output sets of transactions in consecutive decided intervals starting with I_0 implements an order-fair SMR.*

Proof. The agreement property of the consensus protocol ensures that each correct process outputs the same set of transactions for each interval. The fact that output transactions come for consecutive intervals, and starting with the first interval, guarantees the stability of the transactions that are output. The order-fairness property comes from Lemma 6.1 and the fact that the leader must collect submissions from at least $2f + 1$ processes (Lemma 6.2). □

6.5 Leaderless Aion

In this section, we present Leaderless Aion, an order-fair implementation of an SMR, by combining the ordering step (Section 6.3) with a leaderless consensus protocol.

6.5.1 Leaderless Consensus

Without a leader, agreeing on a set of transactions for an interval is not straightforward. For instance, two correct processes may submit two sets of transactions of equal size that only differ by one transaction. To achieve consensus without a leader, [137] relies on an *adopt-commit* object. Intuitively, an adopt-commit object enables processes to adopt the highest value that they witness and later commit to this value once enough processes have adopted it. Thus, the consensus algorithm uses consecutive rounds where processes converge towards the highest value. For two sets of transactions, we define the highest value as the largest set. In case of a tie, two sets can be sorted deterministically using a lexicographical order.

6.5.2 Leaderless SMR Protocol

Leaderless Aion comprises the ordering phase that is executed continuously by processes and a decision phase for each interval. The decision phase consists of an exchange step followed by a consensus step. To preserve the guarantees of the ordering phase (Lemma 6.2), processes first exchange their sets of ordered transactions before executing the consensus protocol.

Algorithm 6.5 presents our leaderless algorithm for order-fair SMR. First, when a process observes that the interval I_k can be decided (line 5), it broadcasts the set of transactions that it has ordered for I_k (line 7). Then, once a process has received the sets of at least $2f + 1$ processes (line 12), it joins the consensus instance for the interval I_k (line 13). The exchange step ensures that the value that it broadcasts satisfies the external validity property (cf. Section 6.1.6).

Theorem 6.2. *A protocol using Algorithm 6.5 to output sets of transactions in consecutive decided intervals starting with I_0 implements an order-fair SMR.*

Algorithm 6.5 Leaderless Aion

State
  $C \leftarrow [\mathbb{N} \rightarrow []]$ ▷ *transactions ordered for each interval*
  $E \leftarrow [\mathbb{N} \rightarrow []]$ ▷ *sets received for each interval*
  $nextSub \leftarrow 0$ ▷ *next interval submitted by* p_i

upon $\mathsf{Clock}_i() \geq (k+1)\ell + 3\Delta$ **do** ▷ *I_k can be decided*
  $\langle C_k \rangle \leftarrow \mathsf{DS.Sign}(sk_i, C[k])$ ▷ *sign set ordered by p_i for I_k*
  $\mathsf{Broadcast}(\text{EXCHANGE}, \langle C_k \rangle)$ ▷ *broadcast ordered set*
  $nextSub \leftarrow nextSub + 1$

upon receiving a message $(\text{EXCHANGE}, C_k)$ from p_j **do**
  if $\mathsf{DS.Verify}(pk_j, C_k)$ **then** ▷ *verify signature*
    $E[k] \leftarrow E[k] \cup C_k$ ▷ *store set of p_j*
    if $|E[k]| \geq 2f + 1$ **then**
      $\mathsf{Leaderless\text{-}Propose}(k, E[k])$ ▷ *propose $E[k]$ for I_k*

Proof. The proof is analogous to the leader-based case. It results directly from the agreement property of the consensus protocol combined with the stability of the transactions that are output, Lemma 6.1, and Lemma 6.2. □

6.6 Discussion

6.6.1 Comparison to Pompē and Aequitas

In Aequitas [12], a transaction t_1 must be ordered before a transaction t_2 if a predetermined proportion of processes have observed t_1 before t_2. The ordering paradigm of Pompē [13] is strictly weaker than that of Aequitas. In Pompē, to require that t_1 be ordered before t_2, it is not sufficient that all processes observe t_1 before t_2; it must also be that all correct processes observe t_1 before any of them observe t_2. Nevertheless, Pompē is an attractive trade-off because, on the one hand, it does not require building graphs of potentially cyclic dependencies between transactions, and on the other hand, clients can send their transactions to a single process.

In Aequitas, although the paradigm is fairer than Pompē, a client still has to send its transactions to all processes, and thus, clients can also be disadvantaged based on their distances to the set of all processes. By assigning timestamps to transactions, we lean towards Pompē's paradigm, which is more scalable [16, 13, 2]. However, instead of computing a timestamp using the median value of the timestamps observed by processes, we compute the send timestamp of a single process. This enables clients to choose the processes they send their transactions to. The results in [18] rely on differential validity for consensus [94] and show that when $f = \lceil \frac{n}{3} \rceil - 1$, for two transactions t_1 and t_2, if a single correct process observes t_2 before t_1, then it cannot be required from any protocol to output t_1 before t_2. By leveraging secure timestamping and allowing clients to choose a single process, a client can select the process it is the closest to.

This diminishes the influence of network delays, both between clients and processes and between processes, and thus reduces the fairness gap between the relative ordering paradigm and the absolute ordering paradigm.

6.6.2 Byzantine Behaviours and Asynchrony

The use of TEEs prevents Byzantine processes from lying about the values of their clocks or from using the values of another clock. Byzantine processes may still introduce biases in the measurements of network delays. First, they can try to beat the network and reduce network delays by using the lack of triangle inequality in network delays [14]. Byzantine processes may also induce longer network delays by simply retaining messages. In both cases, biases are handled by verifying network delays: if a Byzantine process introduces a bias, it has to commit to that bias and, therefore, cannot take advantage of it. Actual variations in network delays are taken into account by the Network Challenge protocol. If a process detects a change in its network delays, it can impose a cooldown period on impacted processes before starting to validate their transactions again. The cooldown period allows pending transactions from other processes to be committed before using new values of network delays.

During asynchronous periods or in the presence of an adversary controlling the network, the network delays are unknown, and the clocks can become desynchronized. In this case, our protocols lose liveness but maintain safety. An increase in the offsets between the clocks of processes only diminishes the level of fairness in the ordering of the transactions that are output. Finally, various attacks have been identified against the security of TEEs [138, 139] but are out of the scope of this dissertation.

6.7 Conclusion

In this chapter, we presented Aion, a set of protocols that enable a secure ordering of transactions. Essentially, Aion protocols do not require clients to broadcast their transactions to all processes and do not disadvantage processes based on their network delays to other processes. Aion leverages TEEs to help determine the times when transactions are broadcast, and uses this information to realise leader-based and leaderless SMR protocols with an accurate ordering of transactions. During synchronous periods, ordering transactions based on the time when they are broadcast reduces to removing network delays between processes. This not only improves order-fairness, but also helps mitigate reordering attacks. For instance, once a transaction t has been received, the timestamp of t resides in the past and thus, no Byzantine process can obtain an earlier timestamp in order to execute a front-running or a sandwich attack.

Chapter 7

Beyond Ordering Linearizability

Despite its ingenuity, ordering linearizability, as mentioned in Chapter 5, weakens the validity property of atomic broadcast. Validity in atomic broadcast [114] requires that a transaction submitted by a correct process is eventually committed. But in Pompē, a transaction t whose timestamps have expired cannot be output, and t must either be discarded or its issuer must collect a new set of timestamps, thus effectively resubmitting t to the SMR protocol. As a result, the validity property can be broken by an asynchronous network that causes the expiration of collected sets of timestamps.

In Chapter 5, we fixed the validity property and ensured that a transaction submitted by a correct process is eventually committed. However, in Chapter 5, for two transactions t_1 and t_2 such as $\mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2)$, t_2 can still be executed before t_1 if t_1 is ordered in a later epoch than t_2, and this results in a weaker safety property that if t_1 was ordered before t_2 simply based on $\mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2)$. A stronger ordering paradigm, fair separability, is presented in [98] and [16]. Fair separability is based on the same observation on ranks that ordering linearizability, i.e., $\mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2)$, but fair separability not only preserves the validity property of atomic broadcast, it also strengthens the safety property of ordering linearizability by ensuring that $\mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2)$ not only guarantees that both t_1 and t_2 are committed, but also that t_1 is committed before t_2.

SMRFS [20] is the first and only known implementation of fair separability, and, furthermore, it also achieves resilience to downgrade attacks. However, to do so, processes must rebroadcast each transaction they observe. As a result, SMRFS outputs every transaction observed by any correct process, even if it was sent by a Byzantine process to a single correct process. Unfortunately, this approach enables a Byzantine process to ensure that each of its transactions t is output by only sending t to a single correct process, and thus at a constant cost of $\mathcal{O}(1)$ network resources. In comparison, a correct process broadcasts its transactions at a cost of $\mathcal{O}(n)$ network resources. If both correct and Byzantine processes have equal networking capabilities, SMRFS becomes vulnerable to attacks on chain quality, where transactions sent by Byzantine processes dominate the output.

It has remained an open problem whether the approach in [20] is necessary for fair separability or if a protocol for fair separability could be devised that does not require to output every transaction submitted by a Byzantine process. The crux of the problem resides in the

fact that when a correct process p_i has assigned a sequence number s to a transaction t, then s could eventually be used as the median value of a set of $2f + 1$ sequence numbers for t, and thus t and s must somehow be taken into consideration when deciding the output of the SMR. To solve this, our protocol first builds on the observation in [20] that when fair separability requires that a transaction t_1 be ordered before a transaction t_2, then any set of $n - f$ distinct processes contains a set P_1 of at least $f + 1$ correct processes, and the $f + 1^{th}$ value of the sequence numbers assigned to t_1 by processes in P_1 is lower than the median value of any set of $2f + 1$ sequence numbers assigned to t_2. Then, we leverage the commit protocol presented in [2] to define commit conditions that do not violate fair separability. Finally, we integrate these ideas into an SMR protocol that ensures fair separability.

To ease the understanding of our algorithm, we first present in Section 7.3 a safe version of our algorithm that may lose liveness in certain scenarios. We then show how to fix liveness in Section 7.4 in order to satisfy both the safety and liveness properties of SMR. More formally, our contribution is twofold.

- We present an implementation of fair separability that preserves chain quality: submitting a transaction has the same asymptotic cost for all processes, and the output requires input from a quorum of processes.
- We show that fair separability can be achieved without having to output every transaction observed by a correct process, which is of independent theoretical interest.

Chapter Outline. The rest of this chapter is structured as follows. In Section 7.1, we introduce the building blocks of the protocol presented in this chapter. We begin by presenting a safe implementation of fair separability in Section 7.3. We then address liveness in our protocol in Section 7.4. Section 7.5 presents our results and their associated proofs. Finally, we summarise the chapter and conclude in Section 7.6.

7.1 Preliminaries

7.1.1 Sequence Numbers

In this chapter, processes use the SeqNum implementation of the Rank function, i.e., Rank = SeqNum. Each process p_i has a local sequence number SeqNum_i that is used to assign ordering indicators to transactions. Whenever p_i observes a transaction t, it assigns the value of SeqNum_i to t and increments SeqNum_i.

7.1.2 Secure Broadcast

The secure broadcast protocol is a multi-shot version of the reliable broadcast protocol [113] where for each index k, a process p_i *secure-broadcasts* a value v, and all correct processes *secure-deliver* v for the index k of p_i.

Definition 7.1 (Secure Broadcast Problem)**.** *A secure broadcast protocol ensures the following properties.*

- ***SB-Validity***. *If a correct process p_i secure-broadcasts a value v for its index k, then every correct process secure-delivers v for the index k of p_i.*
- ***SB-Integrity***. *If a correct process p_j secure-delivers a value v for the index k of p_i and that p_i is correct, then p_i has secure-broadcast v.*
- ***SB-Agreement***. *If a correct process secure-delivers the value v for the index k of p_i, then every correct process eventually secure-delivers v for the index k of p_i, and no correct process secure-delivers for the index k of p_i a value v' such that $v' \neq v$.*

Note that in the definition of secure broadcast, a process is not required to use consecutive values of indices, and thus, secure broadcast is different than FIFO broadcast [30]. Our algorithm in Section 7.4 leverages secure broadcast to build proofs that guarantee that for an index k, a process has secure-broadcast a value for all the previous values $k' \leq k$.

7.1.3 Byzantine Agreement

A leader-based Byzantine agreement protocol [7] enables all correct processes to agree on a unique value proposed by a leader in the presence of Byzantine processes. We further require that the decided value satisfies an *external validity* predicate [25]. In this chapter, we define the predicate γ that holds for any output (V) that contains inputs from at least $2f+1$ distinct processes, each accompanied by a valid signature. More formally,

$$\gamma \colon V \mapsto |V| \geq 2f+1 \ \wedge \ \forall \ (i, v_i, \sigma) \in V, \mathsf{DS.Verify}(pk_i, \sigma, v_i) = 1.$$

Definition 7.2 (Byzantine Agreement Problem)**.** *A Byzantine agreement protocol ensures the following properties.*

- ***BA-Termination***. *Each correct process eventually outputs V.*
- ***BA-Agreement***. *All correct processes output the same V.*
- ***BA-External-Validity***. *If a correct process outputs V, then $\gamma(V)$ holds.*

Throughout this chapter, the leader initializes a consensus instance id with input value v by employing $\mathsf{Consensus\text{-}Propose}(id, v)$.

7.1.4 State Machine Replication

As defined in Section 2.2.5, SMR enables correct processes to agree on a total ordering of transactions. In this chapter, the output of the SMR is divided into logical epochs. Processes start with epoch 1 and use agreement to output the set of transactions in each consecutive epoch. A correct process p_i delivers the transactions decided in an epoch e only if p_i has already delivered the transactions in every epoch $e' < e$. Furthermore, our protocol relies on the ordering paradigm introduced in [13] whereby each transaction $t \in \mathcal{T}$ is output with a sequence number $s \in \mathbb{N}$. Consequently, transactions output during an epoch are ordered based on their respective sequence numbers. Let t_1 and t_2 represent two transactions output during an epoch with their respective sequence numbers s_1 and s_2. If $s_1 < s_2$, then $t_1 \prec t_2$.

7.2 Fair Separability

The term fair separability was introduced in [16] to denote the strengthened version of ordering linearizability introduced in [98]. Ordering linearizability extends the SMR specification by adding constraints to the ordering of committed transactions. However, to preserve liveness, ordering linearizability weakens validity by allowing the SMR protocol to drop transactions submitted by correct processes, and the safety property, expressed as a condition on the ordering of the transactions output, can vanish in case of an asynchronous network. These shortcomings are expressed implicitly in the definition of ordering linearizability (cf. Definition 2.3) whereby the ordering constraint only applies to committed transactions, and without a validity property guaranteeing that transactions submitted by correct processes are eventually committed by the protocol. On the other hand, for two transactions t_1 and t_2, fair separability applies unconditionally based simply on the observation that $\mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2)$.

Definition 7.3 (Fair Separability)**.** $\forall\ t_1, t_2 \in \mathcal{T}$,

$$\mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2) \Rightarrow t_1 \prec t_2.$$

Consider two transactions t_1 and t_2. First, note that for both ordering linearizability and fair separability, an ordering constraint is expressed between t_1 and t_2 only if all correct processes have assigned an ordering indicator to both t_1 and t_2. This is because with f Byzantine processes any ordering indicator assigned by a correct process could potentially be used as the median of a set of $2f + 1$ values. Then, assume that $\mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2)$.

- **Ordering linearizability.** In Pompē, if the network behaves asynchronously and that the ordering indicators collected for t_1 expire before t_1 is ordered, then t_1 must either be discarded, or its issuer must resubmit t_1 in order to collect a new set of timestamps. The protocol in Chapter 5 ensures that in the same scenario t_1 is not discarded. However, t_1 may still be committed after t_2.

- **Fair separability.** The protocol presented in this chapter ensures both validity and ordering safety by ensuring that both t_1 and t_2 are committed (cf. liveness proof in Theorem 7.1), and that t_1 is committed before t_2 (cf. Theorem 7.2).

In [20], to ensure fair separability, all correct processes eventually assign a sequence number and output a transaction t even if t was only sent by its Byzantine issuer to a single correct process. In this chapter, we show that to ensure fair separability, it is not necessary to output every transaction observed by a few correct processes.

7.3 Safe Implementation

In this section, we present our protocol for SMR with fair separability. To simplify the algorithms, we omit all the cryptographic verifications and assume that correct processes systematically verify those and discard invalid messages.

7.3.1 Overview

Like Pompē, our protocol starts with an ordering phase that is followed by a consensus phase. However, in our protocol, processes need to exchange additional information, such as their sets of pending transactions, in order to achieve fair separability. Furthermore, after the consensus step, an additional delivery step ensures that transactions are committed in a way that preserves fair separability. Our protocol proceeds in consecutive epochs. All correct processes start at epoch 1 and output a set of transactions for the current epoch before proceeding to the next epoch. The set of transactions output during an epoch is appended to the SMR Log. Each epoch comprises the three following steps.

1. *Ordering.* Processes assign sequence numbers to the transactions that they receive and secure broadcast the sequence numbers that they have assigned. When a process secure-delivers $2f + 1$ sequence numbers for the same transaction t, it *orders* t.

2. *Consensus.* Processes agree on a tentative set of transactions to be output during the epoch. Each decided transaction is associated with a sequence number that is the median value of $2f + 1$ sequence numbers.

3. *Delivery.* Processes select among the previously decided set the transactions that can be committed without violating fair separability.

7.3.2 Ordering Step

The ordering step is presented in Algorithm 7.1. The goal of this step is to collect sequence numbers for transactions to facilitate the ordering of these transactions during the delivery phase based on their assigned sequence numbers. A process submits a transaction t to the protocol by broadcasting (SUBMIT, t) (line 12). When a process p_i receives t, p_i assigns a sequence number to t and adds t to its set of pending transactions (line 18). The set of pending transactions keeps track of the sequence numbers assigned by correct processes. This is done to preserve fair separability because the median value of a set of $2f + 1$ sequence numbers could have been assigned by any correct process. Process p_i then secure-broadcasts t and the signed sequence number that it has assigned to t (line 20). This step enables all correct processes to have a consistent view of the sequence numbers assigned by processes. When a correct process p_i has secure-delivered at least $2f + 1$ signed sequence numbers for a transaction t, p_i adds t and its associated sequence numbers to its set of ordered transactions (line 25).

To ensure fair separability, it is sufficient that the output of each epoch contains the union of the sets of pending and ordered transactions of at least $2f + 1$ processes (cf. Theorem 7.2). However, doing so naively would make the protocol vulnerable to an arbitrary increase in its communication complexity. If a Byzantine process were to send a transaction t to less than $f + 1$ correct processes, t would be broadcast as part of their pending sets but could never be output. Transactions sent in this way could indefinitely increase the sizes of the messages sent by processes to report their pending transactions. We thus devise a scheme that enables correct processes to notify of their pending transactions using a constant size. Intuitively, the pending

Algorithm 7.1 Ordering step at process p_i

1: **State**
2: $\mathsf{SeqNum}_i \leftarrow 1$ ▷ *local sequence number of* p_i
3: $seqNumSet : [\mathcal{T} \to []] \leftarrow \emptyset$ ▷ *sequence numbers collected for each tx*
4: $delivered : [P \to [\mathbb{N} \times \{0,1\}^\lambda]]$ ▷ *deliveries from secure-broadcast*
5: $pending : [\mathcal{T} \times \mathcal{S} \times \{0,1\}^\lambda] \leftarrow \emptyset$ ▷ *pending txs*
6: $ordered : [\mathcal{T} \times [\mathcal{S} \times \{0,1\}^\lambda]^{2f+1}] \leftarrow \emptyset$ ▷ *ordered txs*
7: $ackedDeliveries : [P \to [\mathbb{N} \times \{0,1\}]]$ ▷ *shares sent to ack deliveries*
8: $ackShares : [\mathbb{N} \to [\{0,1\}^\lambda]^{f+1}]$ ▷ *shares for history proofs*
9: $indexProof : [\mathbb{N} \to \{0,1\}^\lambda] \leftarrow \emptyset$ ▷ *secure-broadcast delivery proofs*
10: $witnessedTxs : [\mathcal{T}]^*$ ▷ *list of witnessed transactions*

11: **function** SUBMIT(t)
12: $\mathsf{Broadcast}(\text{SUBMIT}, t)$ ▷ *submit transaction* t *to the protocol*

13: **upon** receiving (SUBMIT, t) **do**
14: **if** $t \notin witnessedTxs$ **then**
15: $s \leftarrow \mathsf{SeqNum}_i$ ▷ *assign sequence number* s *to* t
16: $\mathsf{SeqNum}_i \leftarrow \mathsf{SeqNum}_i + 1$ ▷ *increment* SeqNum_i
17: $\sigma \leftarrow \mathsf{DS.Sign}(sk_i, i \| \mathsf{h}(t) \| s)$ ▷ *sign* s
18: $pending \leftarrow pending \cup (t, s, \sigma)$ ▷ *mark as pending*
19: $witnessedTxs \leftarrow witnessedTxs \cup t$ ▷ *marked* t *as witnessed*
20: $\mathsf{SecureBroadcast}(s, (t, s, \sigma))$ ▷ *secure-broadcast* s *for* t *once*

21: **upon** secure-delivering $(s, (t, s, \sigma))$ from p_j **do**
22: $delivered[j][s] = (t, s, \sigma)$ ▷ *update history of* p_j
23: $seqNumSet[t] \leftarrow seqNumSet[t] \cup (s, \sigma)$ ▷ *collect sequence numbers for* t
24: **if** $|seqNumSet[t]| \geq 2f + 1$ and $t \neq \perp$ **then**
25: $ordered \leftarrow ordered \cup (t, seqNumSet[t])$ ▷ *order* t *and the collected set*
26: $k \leftarrow \max_{\forall x \in [1,\ell], delivered[j][x] \neq \perp} (\ell)$ ▷ *longest continuous delivery from* p_j
27: **for** $x \in [1, k]$ **do**
28: **if** $ackedDeliveries[j][x] = 0$ **then** ▷ *index* x *of* p_j *not acked*
29: $\pi \leftarrow \mathsf{TS.SignShare}(tsk_i, (j, x))$ ▷ *create share for index* x *of* p_j
30: $ackedDeliveries[j][x] = 1$ ▷ *mark as acknowledged*
31: $\mathsf{Send}(\text{INDEX}, (j, x, \pi))$ to p_j

32: **upon** receiving (INDEX, (i, k, π)) from p_j **do**
33: $ackShares[k][j] \leftarrow \pi$ ▷ *collect ack shares for index* k *of* p_i
34: **if** $|ackShares[k]| \geq f + 1$ **then** ▷ *ack by at least one correct process*
35: $\Pi \leftarrow \mathsf{TS.Combine}(\{tpk\}, ackShares[k], (i, k))$ ▷ *build history proof*
36: $indexProof[k] \leftarrow \Pi$ ▷ *store proof for* p_i*'s continuous history*

set of a process p_j is the set of transactions associated with the sequence numbers secure-broadcast by p_j for the transactions observed by p_j minus the transactions already committed. More specifically, when a correct process p_i secure-delivers the sequence number s from p_j, it computes the index k (line 26) of the highest sequence number for uninterrupted secure-deliveries from p_j (i.e., $\forall\ k' \leq k, delivered[j][k'] \neq \perp$). Process p_i then send to p_j a threshold signature for index k (line 31). By collecting these shares, p_i can build a proof (line 36) for

an uninterrupted history of assigned sequence numbers, where each index is guaranteed to be secure-delivered due to the SB-Agreement property.

7.3.3 Consensus Step

The consensus step is detailed in Algorithm 7.2. When a process p_i is the leader of an epoch e and its set of ordered transactions is not empty, p_i broadcasts a message (COLLECT, e) to collect sets of pending and ordered transactions from processes (line 42). Upon receiving a COLLECT message, a process p_i responds to the leader with the current value of its local sequence number SeqNum_i and its set of pending and ordered transactions (line 53). However, in lieu of directly sending its set of pending transactions, process p_i sends to the leader only the highest value *maxPending* of the sequence numbers present in its set of pending transactions, associated with a proof *indexProof*[*maxPending*] that p_i has secure-broadcast a value for each index up to *maxPending*. Intuitively, the proof ensures that at least one correct process has delivered a value for each index up to *maxPending*, and the SB-Agreement property ensures that, therefore, every correct process will eventually secure-deliver the same values. A formal proof is deferred to Section 7.5.

When the leader has collected the submissions from at least $n - f$ processes, and the timer has expired, it combines them into its proposal for epoch e and initiates an instance of the Byzantine agreement protocol to decide the tentative set of transactions to be output during epoch e. The actual set of committed transactions is decided during the delivery step.

Remark: throughout this chapter, we treat the consensus protocol as a black box, implying that any partially synchronous consensus protocol can be integrated into our framework. Although we use a leader-based partially synchronous consensus protocol in Algorithm 7.2, it is worth noting that any leaderless partially synchronous consensus protocol also fits within our framework, with all correct processes considered as leaders.

7.3.4 Delivery Step

The delivery step is presented in Algorithm 7.3. Once the result of the consensus instance for epoch e is known, processes locally compute a set of transactions that can be committed without violating fair separability. To this end, each process p_i must (1) remove from the tentative set unsafe transactions for which there may be transactions that could be assigned lower sequence numbers and (2) add transactions that must be ordered before according to fair separability but have not been ordered.

Removing unsafe transactions. The median value associated with a transaction could have been assigned by any correct process. To ensure that no transaction can be issued a median value that is less than the sequence numbers of the transactions in the tentative set, processes use the values of the local sequence numbers SeqNum included in the decided proposal for e. Specifically, they compute the value of *lockedIndex* corresponding to the lowest value of SeqNum in the decided proposal (line 62). Only transactions whose final sequence numbers are less than *lockedIndex* can be committed in the current epoch (line 73). Note that Byzantine processes could try to prevent transactions from being committed by sending superficially low values

Algorithm 7.2 Consensus step at process p_i

37: **State**
38: $epoch \leftarrow 1$ ▷ *consensus epoch*
39: $proposal : [\mathbb{N} \rightarrow []]$ ▷ *leader proposals for epochs*

40: **upon** $|ordered| \geq 1 \ \wedge \ i \equiv epoch \pmod n$ **do** ▷ *p_i is leader of epoch*
41: **wait** epoch $e-1$ is decided **do** ▷ *decide epochs successively*
42: $\mathsf{Broadcast}(\text{COLLECT}, e)$ ▷ *request ordered and pending sets*
43: $\mathsf{Timer}(2\Delta)$ ▷ *set timer for 2Δ*

44: **upon** receiving $(\text{COLLECT}, e)$ from p_j **do**
45: **if** p_j is the leader of epoch e **then** ▷ *$j \equiv e \pmod n$*
46: $maxPending \leftarrow \max_{(t,s,\sigma) \in pending}(s)$ ▷ *latest pending local transaction*

47: **wait** $indexProof[maxPending] \neq \perp$ **do** ▷ *proof for index maxPending*
48: $L \leftarrow \{\}$ ▷ *initialize local submission*
49: $L.ordered \leftarrow ordered$
50: $L.maxPending \leftarrow maxPending$
51: $L.pendingProof \leftarrow indexProof[maxPending]$
52: $\sigma \leftarrow \mathsf{DS.Sign}(sk_i, \mathsf{h}(i \| e \| L))$ ▷ *sign local submission L*
53: $\mathsf{Send}(\text{LOCAL}, (e, L, \sigma, \mathsf{SeqNum}_i))$ to p_j

54: **upon** receiving $(\text{LOCAL}, (e, L, \sigma, \mathsf{SeqNum}_j))$ from p_j **do**
55: $proposal[e] \leftarrow proposal[e] \cup (j, L, \sigma, \mathsf{SeqNum}_j)$ ▷ *store p_j's submission*
56: **if** $|proposal[e]| \geq n - f \wedge \mathsf{Timer}$ has expired **then** ▷ *collected $n-f$*
57: $\mathsf{Consensus\text{-}Propose}(e, proposal[e])$ ▷ *submit proposal to consensus*

of SeqNum. To thwart this behaviour, the leader first waits for the expiration of the timer when collecting submissions for its proposal. This ensures that during synchronous periods, the leader receives submissions from all correct processes. Then, the value of *lockedIndex* is computed using only the highest $2f+1$ values of SeqNum in the proposal (line 61).

Adding non-ordered transactions. To ensure that the set of committed transactions does not exclude any transaction that should be ordered before the transactions in the tentative set, each process p_i looks at the pending transactions in the histories of sequence numbers assigned by processes. Specifically, p_i adds to the tentative set any transaction that is not in the ordered sets, but that is present in the pending sets of at least $f+1$ processes (line 71).

Finally, after removing unsafe transactions and adding non-ordered transactions, processes can commit the resulting set C of transactions (line 76).

Algorithm 7.3 Delivery step at process p_i

58: **State**
59: $\mathsf{Log}_i \leftarrow []$ ▷ *log of committed transactions*

60: **upon** deciding $proposal[e]$ for epoch e **do**
61: $M \leftarrow \underset{V \subseteq proposal[e], |V|=2f+1}{\arg\max} \sum_{(j,L,\sigma,\mathsf{SeqNum}_j) \in V} (\mathsf{SeqNum}_j)$ ▷ *$2f+1$ highest*
62: $lockedIndex \leftarrow \min_{s \in M}(s)$ ▷ *compute locked index*
63: $O \leftarrow []$ ▷ *ordered sets decided*
64: $P \leftarrow []$ ▷ *pending sets decided*
65: **for** $(j, L, \sigma, \mathsf{SeqNum}_j) \in proposal[e]$ **do**
66: **wait** $(L.maxPending, *)$ secure-delivered from p_j **do**
67: $O \leftarrow O \cup L.ordered$ ▷ *combine ordered sets*
68: **for** $(t, s, \sigma) \in \{delivered[j][s] : s \leq L.maxPending\}$ **do**
69: **if** $(t, s, \sigma) \notin \mathsf{Log}_i$ **then**
70: $P[t] \leftarrow P[t] \cup s$ ▷ *combine pending sets grouped by transaction*

71: $P \leftarrow P \setminus \{P[t] \in P : |P[t]| < f + 1\}$ ▷ *txs that appear in at least $f+1$ sets*
72: $D \leftarrow O \cup P$ ▷ *all decided transactions*
73: $C \leftarrow \{(t, \mathsf{Median}(\{s\}) : (t, \{s\}) \in D \wedge \mathsf{Median}(\{s\}) \leq lockedIndex\}$
74: $ordered \leftarrow ordered \setminus C$ ▷ *update ordered set*
75: $pending \leftarrow pending \setminus C$ ▷ *update pending set*
76: $\mathsf{Log}_i.\mathsf{append}(C)$ ▷ *commit set C*
77: $\mathsf{Update}(D)$ ▷ *required for liveness (see Section 7.4)*

7.4 Fixing Liveness

In this section, we address liveness issues with Algorithm 7.3.

7.4.1 Issue with the Previous Protocol

Our previous implementation ensures fair separability (Theorem 7.2), but it does not ensure liveness. Consider the following scenario with $P = \{p_1, p_2, p_3, p_4\}$, and where p_1 is Byzantine.

- Correct processes p_2, p_3, p_4 have local sequence number $2, 4, 8$, respectively, and $\mathsf{SeqNum}_1 = 9$.
- A transaction t is ordered with the set of sequence numbers $S = [4, 8, 9]$, and thus output in an epoch e with a sequence number $\overline{s} = \mathsf{Median}(S) = 8$.
- After receiving t, the local sequence numbers SeqNum of processes (p_2, p_3, p_4) are $(2+1, 4+1, 8+1) = (3, 5, 9)$, respectively. If Byzantine process p_1 does not send any response to the leader requests for collection, and no other transaction is submitted, then the computed value of $lockedIndex$ is $\min([3, 5, 9]) = 3 < 8$.

Algorithm 7.4 Update of SeqNum_i at process p_i

```
78: function Update(D)
79:     s_max ← max_{(t,s̄)∈D} (s̄)                    ▷ highest decided Median
80:     lastSeq ← SeqNum_i
81:     SeqNum_i ← max(SeqNum_i, s_max)               ▷ update SeqNum_i
82:     if SeqNum_i − lastSeq > 0 then
83:         for k ∈ [lastSeq, SeqNum_i) do            ▷ fill the gap
84:             SecureBroadcast(k, ⊥)
```

If no other transaction is submitted, and that Byzantine process p_1 remains silent, then t cannot be committed. To ensure liveness, we modify the previous protocol. Intuitively, processes increase the values of their local sequence numbers so that the transactions decided during consensus, such as t, can be committed in the next epoch.

7.4.2 Fixing Liveness

To ensure liveness, we must guarantee that a transaction broadcast by a correct process is eventually committed. To this end, upon deciding a set of transactions for an epoch e, processes increase the value of their local sequence numbers to the highest sequence number in the set of decided transactions. We implement this increase of sequence numbers by adding in Algorithm 7.3 a call to the Update function (line 77). The Update function is detailed in Algorithm 7.4. First, recall that in Algorithm 7.3, two sets of transactions are computed:

- a tentative set D (line 72) consisting of all the decided transactions that could be committed,
- a committed set C (line 73) consisting of the transactions that can be committed without violating fair separability.

In Algorithm 7.4, a process p_i computes the highest sequence number s_{max} in the set D, and uses s_{max} to update its local sequence number SeqNum_i. This ensures that when the network becomes synchronous (i.e., after GST), then all the transactions decided in an epoch e can be committed in epoch $e + 1$. The proof of liveness is presented in Theorem 7.1.

7.5 Protocol Analysis

In this section, we prove that our protocol implements an SMR protocol with fair separability and discuss its chain quality.

7.5.1 State Machine Replication

Lemma 7.1 (History Liveness)**.** *If a correct process p_i secure-broadcasts a value v for its index k, then p_i eventually receives enough acknowledgment shares to build a valid proof Π for its index k.*

Proof. The SB-Validity property of Secure Broadcast (cf. Definition 7.1) ensures that every correct process eventually secure-delivers v. Because p_i is correct, it also has previously secure-broadcast a value $v_{k'}$ for each index $k' < k$. Hence, every correct process eventually secure-delivers a value for each index $k' \leq k$ and sends to p_i an acknowledgment share for index k (line 31). As a result, p_i receives at least $n - f \geq f + 1$ shares and can combine them to build a valid proof for its index k (line 36). □

Lemma 7.2 (History Consistency)**.** *If a correct process receives a valid index proof* Π *for the index* k *of* p_j*, then every correct process eventually secure-delivers some value* $v_{k'}$ *for each index* $k' \leq k$ *of* p_j*, (i.e.,* $delivered[j][k'] \neq \perp$*), and for each index* k' *of* p_j*, every correct process secure-delivers the same value* $v_{k'}$*.*

Proof. In order to build a proof Π for its index k, a process p_j must combine $f + 1$ acknowledgment shares for the index k. Consequently, at least one correct process p_i has created a share π for the index k of p_j. Process p_i only generates π if it has secure-delivered from p_j a value $v_{k'}$ for every index $k' \leq k$. As a result, from the SB-Agreement property of Secure Broadcast (cf. Definition 7.1), every correct process also delivers an identical value $v_{k'}$ for every index $k' \leq k$. □

Lemma 7.3 (Delivery Agreement)**.** *During the delivery step (cf. Algorithm 7.3), each correct process commits the same set of transactions.*

Proof. The BA-Agreement property of consensus (cf. Definition 7.2) ensures that each correct process decides the same proposal for each epoch. A proposal is comprised of submissions from at least $2f + 1$ processes. A submission from a process p_i contains a set of ordered transactions, a value of *maxPending*, a valid proof Π for the index *maxPending* of p_i, and the latest value of SeqNum_i. Based on Lemma 7.2, a valid proof Π for p_i guarantees that all correct processes eventually deliver a consistent view of the history of p_i and that, therefore, the wait at line 66 terminates for each submission in the proposal. It results that all correct processes use a deterministic algorithm based on the same data and, therefore, commit the same set of transactions. □

Theorem 7.1 (SMR)**.** *Our protocol, consisting of Algorithm 7.1, Algorithm 7.2, Algorithm 7.3, and Algorithm 7.4, implements an SMR protocol (cf. Definition 2.1).*

Proof. We prove each property separately.

Safety. Each correct process p_i commits transactions in successive epochs, and for each epoch e, p_i waits until the consensus for epoch $e - 1$ has terminated before taking part in epoch e. Consequently, correct processes commit epochs in the same order, starting with epoch 1, and from Lemma 7.3, we know that each correct process commits the same set of transactions during each epoch.

Liveness. If a correct process broadcasts a transaction t, then each correct process p_i eventually receives t and secure-broadcasts the sequence number that it assigns to t. As a result, every correct process eventually secure-delivers at least $n - f \geq 2f + 1$ correctly signed sequence

numbers for t and adds t to its local *ordered* set. Each decided epoch contains submissions from at least $n - f$ processes and, therefore, submissions from at least $f + 1$ correct processes. Therefore, from then on, the set O (lines 63 and 67) will contain t, and thus t is included in the set D of decided transactions (line 72) of each subsequent epoch. Let S denote the set of sequence numbers associated with t in the decided proposal D. If t is present more than once with different sets, processes deterministically select the set with the lowest median value. Let $\overline{s} = \mathsf{Median}(S)$.

It remains to show that the *lockedIndex* value computed by each correct process (line 62) is eventually greater than $\overline{s}$, and that, therefore, the condition on line 73 is satisfied so that t can be committed. First, note that Algorithm 7.4 ensures that when a set D is decided, all correct processes increase their sequence numbers to a value greater than or equal to $\overline{s}$ (line 81). Then, after GST, a correct leader is eventually elected in an epoch e and waits for 2Δ when collecting submissions. Therefore, the leader of epoch e includes in its proposal the submissions from all correct processes. Finally, note that in epoch e, the *lockedIndex* value is the lowest value among the $2f + 1$ highest values of SeqNum included in the decided proposal (lines 61 and 62). It follows that even if f Byzantine processes were to send superficially low values of SeqNum, the value of *lockedIndex* will still be lower bounded by the sequence number of a correct process that has previously set its local sequence number SeqNum to a value greater than or equal to $\overline{s}$, and t is thus committed in epoch e. □

7.5.2 Fair Separability

Lemma 7.4. *Let $S_I = \{\mathsf{SeqNum}_i(t) : i \in I\}$ denote the set of sequence numbers assigned to t by a set $I \subseteq P$ of distinct processes.*

$$|I| \geq 2f + 1 \Rightarrow \mathsf{MinRank}(t) \leq \mathsf{Median}(S_I) \leq \mathsf{MaxRank}(t)$$

Proof. If $|I| \geq 2f + 1$, then there are in S_I at least f values both before and after $\mathsf{Median}(S_I)$. As a result, $\mathsf{Median}(S_I)$ is both lower-bounded and upper-bounded by the sequence numbers assigned to t by correct processes. □

Lemma 7.5. *Let t_1 and t_2 denote two transactions. Let S_2 denote a set of $2f + 1$ sequence numbers assigned to t_2 by any set of distinct processes. Let S_1 denote a set of $f + 1$ sequence numbers assigned to t_1 by any set of* correct *processes.*

$$\mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2) \Rightarrow \mathsf{Median}(S_1) < \mathsf{Median}(S_2)$$

Proof. From Lemma 7.4, we have for t_2,

$$\mathsf{MinRank}(t_2) \leq \mathsf{Median}(S_2) \leq \mathsf{MaxRank}(t_2).$$

By assumption, S_1 only contains the sequence numbers assigned to t_1 by correct processes. Therefore we have

$$\forall s \in S_1, s \leq \mathsf{MaxRank}(t_1) \Rightarrow \mathsf{Median}(S_1) \leq \mathsf{MaxRank}(t_1).$$

It directly follows that

$$\mathsf{Median}(S_1) \leq \mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2) \leq \mathsf{Median}(s_1).$$

□

Theorem 7.2 (Fair Separability)**.** *An SMR protocol using Algorithm 7.1, Algorithm 7.2, and Algorithm 7.3 to commit transactions in successive epochs ensures fair separability:*

$$\forall\ t_1, t_2 \in \mathcal{T}, \mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2) \Rightarrow t_1 \prec t_2.$$

Proof. Let t_1 and t_2 denote two transactions output in epochs e_1 and e_2, respectively, and such that $\mathsf{MaxRank}(t_1) < \mathsf{MinRank}(t_2)$. First, if $e_1 < e_2$, then $t_1 \prec t_2$ holds trivially. Otherwise, we must show that if t_2 is output during an epoch $e_2 \leq e_1$, then we have $e_2 = e_1$ and t_1 is also output in e_2 with a sequence number lower than t_2.

Assume that t_2 is output during an epoch e_2 with a sequence number $\overline{s}_2$. In e_2, only transactions that have a sequence number that is less than or equal to *lockedIndex* are output (line 73), and therefore we have $\overline{s}_2 \leq$ *lockedIndex*. Furthermore, the BA-External-Validity property of the consensus protocol (cf. Definition 7.2) guarantees that the leader proposal $proposal[e_2]$ for epoch e_2 is based on the submissions of at least $2f + 1 \leq n - f$ distinct processes. Therefore, because *lockedIndex* is the lowest value of SeqNum_i among the collected responses (line 62), *lockedIndex* is less than or equal to the local sequence numbers SeqNum_i of a set P_1 of at least $f + 1$ correct processes.

Due to Lemma 7.1, each correct process can eventually build a proof for each index that it secure-broadcasts. Thus, each correct process $p_i \in P_1$ includes in its submission for $proposal[e_2]$ the value *maxPending* of its highest pending transaction and a valid proof associated with the value of the index *maxPending* of p_i. Lemma 7.2 ensures that during the delivery steps, correct processes can receive all the pending transactions for each submission in $proposal[e_2]$. Recall that the local sequence number SeqNum_i of each correct process $p_i \in P_1$ is greater than or equal to *lockedIndex*. Furthermore, by assumption we have $\mathsf{MaxRank}(t_1) < \overline{s}_2$, and thus $\mathsf{MaxRank}(t_1) < \overline{s}_2 \leq$ *lockedIndex*. Hence, every correct process in P_1 must have already assigned a sequence number to t_1, and thus includes t_1 in its set of pending transactions. And because $|P_1| \geq f + 1$, the condition on line 71 does not exclude t_1, and t_1 is also output in e_2. Finally, using Lemma 7.5, we can conclude that t_1 is output with a sequence number $\overline{s}_1 < \overline{s}_2$. □

7.5.3 Discussion

In SMRFS [20], a Byzantine process can submit a transaction at a cost of $\mathcal{O}(1)$ network resources by sending it to a single correct process that will rebroadcast it to all processes. Simultaneously, for a correct process to submit a transaction, it has to broadcast the transaction to all processes, incurring a cost of $\mathcal{O}(n)$. This enables a greedy Byzantine process to submit a distinct transaction t_i to each process in lieu of broadcasting a unique transaction t. As a result, if each Byzantine process sends a distinct transaction to each correct process, f Byzantine processes can submit $\mathcal{O}(n^2)$ transactions using $\mathcal{O}(n^2)$ network resources. On the other hand, using $\mathcal{O}(n^2)$ network resources, correct processes only submit $\mathcal{O}(n)$ transactions by broadcasting them.

In our protocol, a transaction t can only be output if at least $f + 1$ (i.e., $\mathcal{O}(n)$) processes have assigned a sequence number to t. As a result, whether a Byzantine process sends its transaction t to $f + 1$ correct processes or f Byzantine processes broadcast a sequence number for t, Byzantine processes still need $\mathcal{O}(n)$ network resources to submit a transaction. Finally, the external validity condition ensures that the output of each epoch contains the submissions of at least $n - f$ processes and thus submissions from at least $f + 1$ correct processes. This ensures that the output of an epoch includes the submission of at least one correct process that, in turn, has collected transactions from all processes.

Although typical SMR approaches focus only on improving throughput [56, 58, 60, 61], application of the SMR paradigm to decentralized applications has brought a wide range of applications where it is more important to prioritize the inclusion of inputs from all processes over a high throughput. These applications include collaborative decision-making, consortium blockchains, distributed voting systems, and financial reconciliation systems. In these scenarios, ensuring that inputs from all processes are included is fundamental for maintaining fairness, accuracy, comprehensive data representation, and trust among participants. By combining a balanced cost for transaction submission and a merged output, we ensure an inclusive output for each epoch.

In terms of performance, the overhead of our protocol is limited. Our protocol has a fast path of 9 rounds, which is equivalent to or less than other order-fairness solutions [13, 25, 16, 20]. Our protocol also incurs an $\mathcal{O}(n^3)$ communication complexity, which is identical to previous order-fairness solutions [13, 25], and a linear multiplicative factor increase with respect to Themis [16] and SMRFS [20]. The main computational overhead in our protocol resides in the use of threshold encryption for the sequence numbers broadcast by processes. We defer a comprehensive experimental evaluation of the protocol and a comparison to other solutions for future work.

7.6 Conclusion

In this chapter, we analysed shortcomings both in the formulation of ordering linearizability and in the existing implementation of fair separability. We addressed these shortcomings by introducing an implementation of fair separability that is resilient to attacks on chain quality. Compared to previous work that achieve fair separability by adding a requirement that forces the observation of a transaction by all correct processes, the novelty of our solution resides in the fact that it achieves fair separability while strictly adhering to the prescribed ordering paradigm. As a result, our solution also refines the requirements for achieving fair separability by showing that unconditional ordering linearizability does not require to commit each transaction observed by a correct process.

Chapter 8

Conclusion

In this dissertation, we have presented multiple contributions focused on optimising ordering linearizability in the hope of mitigating attacks and presenting a reinforced case for its application in SMR. Our research surrounded three complementary objectives to address shortcomings in current approaches to ordering linearizability. **Objective 1** focused on minimizing the overhead induced by ordering linearizability. **Objective 2** addressed shortcomings with respect to the essence of order-fairness: resilience to reordering attacks. **Objective 3** strengthened ordering linearizability by solving its network bias and by ensuring that it applies to all transactions. Overall, our goal was to strive towards easing the integration of order-fairness in SMR by reducing its cost, improving its resilience, and strengthening its paradigm.

8.1 Research Outcome

8.1.1 Objective 1: Reduce the Cost of Ordering Linearizability

The application of order-fairness in SMR is currently underwhelming, primarily due to the cost of integrating techniques such as ordering linearizability. To this end, the first research objective of this dissertation focused on reducing those costs to make order-fairness more accessible and attractive. The objective comprised two sub-objectives, namely: (i) reducing latency, and (ii) reducing communication complexity. In Chapter 3, we highlighted that the latency for achieving ordering linearizability could be practically halved. Furthermore, in Chapter 4, we introduced a protocol that achieves ordering linearizability without introducing any message delay and is thus optimal. Chapter 5 presented work that leveraged cryptography to achieve ordering linearizability with optimal communication complexity. The success in achieving our two sub-objectives showed that ordering linearizability could be achieved without introducing additional overhead costs in normal SMR applications, making it a practical strategy for achieving order-fairness.

8.1.2 Objective 2: Improve Resilience to Reordering Attacks

The previous objective revealed that overhead costs introduced for order-fairness could be reduced, however, the techniques were still vulnerable to reordering attacks. The second research

objective, therefore, aimed at improving the resilience of ordering linearizability to reordering attacks. Due to the nature of attacks against existing techniques, this objective required a new protocol to be developed that addressed these concerns and mitigated the shortcomings identified. In Chapter 4, we analysed vulnerabilities to reordering attacks in implementations of time order-fairness and blind order-fairness. We then devised a protocol that addressed these vulnerabilities and applied our findings in Chapter 5. Although the protocol presented in Chapter 6 primarily aimed at strengthening ordering linearizability, it also improved resilience to reordering attacks by sequencing transactions at the time they are sent. By analysing the various shortcomings present in implementations of order-fairness, we have improved its resilience to reordering attacks by addressing multiple vulnerabilities, achieving our objective.

8.1.3 Objective 3: Strengthen Ordering Linearizability

The third and final research objective was to strengthen the ordering paradigm itself. The first sub-objective led us to analyse weaknesses in the formulation of ordering linearizability. In Chapter 6, we analysed the network bias that was intrinsic to ordering linearizability and addressed this bias in a protocol that leveraged TEE. Then, in Chapter 7, to answer our second sub-objective, we devised a protocol that implemented fair separability, the same ordering paradigm as ordering linearizability, but applied to all the transactions. In the end, we have strengthened ordering linearizability by addressing its network bias in Chapter 6 and by extending it to all the transactions in Chapter 7.

8.1.4 Overall Achievement

This dissertation successfully achieved the overarching objective of optimising order-fairness in blockchains using ordering linearizability. By achieving **Objective 1**, we have not only presented the lower bounds for accomplishing ordering linearizability, but we have implemented efficient protocols that attained those bounds, thus showing how to achieve ordering linearizability with virtually no additional overhead costs. **Objective 2** and **Objective 3** ensured that order-fairness did, in fact, meet its goal of preventing adverse reordering by Byzantine processes. In conclusion, we have achieved all of our research objectives focused on optimising ordering linearizability, and as a result, we have made ordering linearizability more applicable to various SMR systems, especially to those in the world of blockchains where the ordering of transactions is crucial.

8.2 Future Work

The work in this dissertation not only presents novel steps forward in order-fairness, but also uncovers potential avenues to explore for further refinement. By using our findings as a foundation for further improvement, there are three immediate directions that could be taken for future research: (i) using formal methods to characterise resilience to reordering attacks, (ii) combining the separate works proposed throughout this dissertation into a unified protocol, and (iii) transposing the order-fairness approach presented in this thesis to permissionless blockchain

environments. These efforts can provide foundations for further application of order-fairness in blockchains while joining the global effort for secure distributed state machine replication with digital financial systems.

8.2.1 Formalisation of Resilience to Reordering Attacks

By combining time order-fairness and blind order-fairness with a leaderless consensus protocol [2], we ensured that our protocol was resilient to harmful reordering attacks. Intuitively, a leaderless protocol ensures that no Byzantine process can censor or omit client transactions. Time order-fairness guarantees a fair ordering of transactions according to their received orders, and blind order-fairness prevents Byzantine processes from issuing transactions based on the content of pending transactions. Future work will focus on formalising resilience to reordering attacks and will investigate impossibility results showing the minimal influence of Byzantine processes with regard to resilience to reordering attacks.

8.2.2 In Search of a Unified Algorithm

In this dissertation, we have shown lower bounds for the latency and the communication complexity of ordering linearizability. In addition, we have improved the resilience of ordering linearizability to reordering attacks by combining it with payload obfuscation. Finally, we have also improved the ordering paradigm by solving the network bias and by implementing fair separability. Future direction will consist of devising a protocol that combines the different properties that have been achieved independently in this dissertation. In particular, when all properties cannot be achieved simultaneously, the research will investigate the various trade-offs and their respective results.

8.2.3 Application to Open Blockchains

The results presented in this dissertation apply to a permissioned setting with a predetermined set of processes and an upper bound on the number of Byzantine processes. However, some of our results could be transposed to the permissionless setting. Future work will focus on applying our results to the context of open blockchains. The transposition of our results would require an approach analogous to [140] that consists of introducing a Sybil-resistant mechanism such as proof-of-work [9] or proof-of-stake [67] in order to achieve a Byzantine resilient ordering.

Notations

Notation	Description
P	The set of all processes
P^{Corr}	The set of correct processes
P^{Byz}	The set of Byzantine processes
n	The total number of processes
f	The bound on the number of Byzantine processes
t	A transaction
$\mathcal{T}$	The set of all possible transactions
c_t	Ciphertext of a transaction t
s	An ordering indicator (either a timestamp or a sequence number)
S	A set of ordering indicators
Rank	Function assigning ordering indicators
$\mathsf{MinRank}$	Function returning the lowest ordering indicator among correct processes
$\mathsf{MaxRank}$	Function returning the highest ordering indicator among correct processes
Clock	Function returning ordering indicators that are timestamps
SeqNum	Function returning ordering indicators that are sequence numbers
h	Hash function
Log_i	The set of transactions committed by process p_i
$t \in \mathsf{Log}$	Transaction t is committed
Δ	Latency bound after GST
ℓ	Bit length of a transaction
λ	Cryptographic security parameter
δ	Maximum offset between the values of clocks
ϵ	Maximum fluctuation of network latencies
$t \sqsubset_i e$	Transaction t is ordered in epoch e by process p_i
$t \sqsubset e$	Transaction t is ordered in epoch e by $2f+1$ processes
$\langle v \rangle_i$	Value v signed by p_i's TEE
$\mathsf{Median}(S)$	$(f+1)^{th}$ value of the set S

Table 8.1: Symbols and Notations.

Acronyms

BFT	Byzantine Fault Tolerance. 9, 11
BOC	Byzantine Ordered Consensus. 37
DAG	Directed Acyclic Graph. 37
DBFT	Democratic Byzantine Fault Tolerance. 19
DLT	Distributed Ledger Technology. 10
GST	Global Stabilization Time. 13, 38
MEV	Maximal Extractable Value. 11, 60
SMR	State Machine Replication. 9, 11, 91
TEE	Trusted Execution Environment. 12, 76, 104

Bibliography

[1] Pouriya Zarbafian and Vincent Gramoli. "Brief Announcement: Ordered Reliable Broadcast and Fast Ordered Byzantine Consensus for Cryptocurrency". In: *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*. Ed. by Seth Gilbert. Vol. 209. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 63:1–63:4. DOI: 10.4230/LIPICS.DISC.2021.63. URL: https://doi.org/10.4230/LIPIcs.DISC.2021.63.

[2] Pouriya Zarbafian and Vincent Gramoli. "Lyra: Fast and Scalable Resilience to Reordering Attacks in Blockchains". In: *IEEE International Parallel and Distributed Processing Symposium, IPDPS 2023, St. Petersburg, FL, USA, May 15-19, 2023*. IEEE, 2023, pp. 929–939. DOI: 10.1109/IPDPS54959.2023.00097. URL: https://doi.org/10.1109/IPDPS54959.2023.00097.

[3] Vincent Gramoli et al. "AOAB: Optimal and Fair Ordering of Financial Transactions". In: *54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2024, Brisbane, Australia, June 24-27, 2024*. IEEE, 2024, pp. 377–388. DOI: 10.1109/DSN58291.2024.00045. URL: https://doi.org/10.1109/DSN58291.2024.00045.

[4] Pouriya Zarbafian and Vincent Gramoli. "Aion: Secure Transaction Ordering Using TEEs". In: *Computer Security - ESORICS 2023 - 28th European Symposium on Research in Computer Security, The Hague, The Netherlands, September 25-29, 2023, Proceedings, Part IV*. Ed. by Gene Tsudik et al. Vol. 14347. Lecture Notes in Computer Science. Springer, 2023, pp. 332–350. DOI: 10.1007/978-3-031-51482-1_17. URL: https://doi.org/10.1007/978-3-031-51482-1%5C_17.

[5] Vincent Gramoli et al. "Resilience to Chain-Quality Attacks in Fair Separability". In: *Computer Security - ESORICS 2024 - 29th European Symposium on Research in Computer Security, Bydgoszcz, Poland, September 16-20, 2024, Proceedings, Part IV*. Ed. by Joaquín García-Alfaro et al. Vol. 14985. Lecture Notes in Computer Science. Springer, 2024, pp. 251–270. DOI: 10.1007/978-3-031-70903-6_13. URL: https://doi.org/10.1007/978-3-031-70903-6%5C_13.

[6] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. [Online] Available: https://bitcoin.org/bitcoin.pdf. 2008.

[7] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. "Reaching Agreement in the Presence of Faults". In: *J. ACM* 27.2 (1980), pp. 228–234. DOI: 10.1145/322186.322188. URL: https://doi.org/10.1145/322186.322188.

[8] John R. Douceur. "The Sybil Attack". In: *Peer-to-Peer Systems, First International Workshop, IPTPS 2002, Cambridge, MA, USA, March 7-8, 2002, Revised Papers*. Ed. by Peter Druschel, M. Frans Kaashoek, and Antony I. T. Rowstron. Vol. 2429. Lecture Notes in Computer Science. Springer, 2002, pp. 251–260. DOI: 10.1007/3-540-45748-8_24. URL: https://doi.org/10.1007/3-540-45748-8%5C_24.

[9] Adam Back et al. *Hashcash-a denial of service counter-measure*. 2013.

[10] Philip Daian et al. "Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 910–927. DOI: 10.1109/SP40000.2020.00040. URL: https://doi.org/10.1109/SP40000.2020.00040.

[11] Gavin Wood et al. "Ethereum: A secure decentralised generalised transaction ledger". In: *Ethereum project yellow paper* 151.2014 (2014), pp. 1–32.

[12] Mahimna Kelkar et al. "Order-Fairness for Byzantine Consensus". In: *Advances in Cryptology - CRYPTO 2020 - 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17-21, 2020, Proceedings, Part III*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12172. Lecture Notes in Computer Science. Springer, 2020, pp. 451–480. DOI: 10.1007/978-3-030-56877-1_16. URL: https://doi.org/10.1007/978-3-030-56877-1%5C_16.

[13] Yunhao Zhang et al. "Byzantine Ordered Consensus without Byzantine Oligarchy". In: *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 633–649. URL: https://www.usenix.org/conference/osdi20/presentation/zhang-yunhao.

[14] Cristian Lumezanu et al. "Triangle Inequality and Routing Policy Violations in the Internet". In: *Passive and Active Network Measurement, 10th International Conference, PAM 2009, Seoul, Korea, April 1-3, 2009. Proceedings*. Ed. by Sue B. Moon, Renata Teixeira, and Steve Uhlig. Vol. 5448. Lecture Notes in Computer Science. Springer, 2009, pp. 45–54. DOI: 10.1007/978-3-642-00975-4_5. URL: https://doi.org/10.1007/978-3-642-00975-4%5C_5.

[15] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. "SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain". In: *Financial Cryptography and Data Security - FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18-22, 2019, Revised Selected Papers*. Ed. by Andrea Bracciali et al. Vol. 11599. Lecture Notes in Computer Science. Springer, 2019, pp. 170–189. DOI: 10.1007/978-3-030-43725-1_13. URL: https://doi.org/10.1007/978-3-030-43725-1%5C_13.

[16] Mahimna Kelkar et al. “Themis: Fast, Strong Order-Fairness in Byzantine Consensus”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. Ed. by Weizhi Meng et al. ACM, 2023, pp. 475–489. DOI: 10.1145/3576915.3616658. URL: https://doi.org/10.1145/3576915.3616658.

[17] Dahlia Malkhi and Pawel Szalachowski. “Maximal Extractable Value (MEV) Protection on a DAG”. In: *4th International Conference on Blockchain Economics, Security and Protocols, Tokenomics 2022, December 12-13, 2022, Paris, France*. Ed. by Yackolley Amoussou-Guenou, Aggelos Kiayias, and Marianne Verdier. Vol. 110. OASIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 6:1–6:17. DOI: 10.4230/OASICS.TOKENOMICS.2022.6. URL: https://doi.org/10.4230/OASIcs.Tokenomics.2022.6.

[18] Christian Cachin et al. “Quick Order Fairness”. In: *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*. Ed. by Ittay Eyal and Juan A. Garay. Vol. 13411. Lecture Notes in Computer Science. Springer, 2022, pp. 316–333. DOI: 10.1007/978-3-031-18283-9_15. URL: https://doi.org/10.1007/978-3-031-18283-9%5C_15.

[19] Danny Dolev and Rüdiger Reischuk. “Bounds on Information Exchange for Byzantine Agreement”. In: *J. ACM* 32.1 (1985), pp. 191–204. DOI: 10.1145/2455.214112. URL: https://doi.org/10.1145/2455.214112.

[20] Vincent Gramoli et al. “Optimal Asynchronous Byzantine Consensus with Fair Separability”. In: *IACR Cryptol. ePrint Arch.* (2024), p. 545. URL: https://eprint.iacr.org/2024/545.

[21] Ittai Abraham et al. “Good-case Latency of Byzantine Broadcast: a Complete Categorization”. In: *PODC ’21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*. Ed. by Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen. ACM, 2021, pp. 331–341. DOI: 10.1145/3465084.3467899. URL: https://doi.org/10.1145/3465084.3467899.

[22] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. “The Byzantine generals problem”. In: *Concurrency: the Works of Leslie Lamport*. Ed. by Dahlia Malkhi. ACM, 2019, pp. 203–226. DOI: 10.1145/3335772.3335936. URL: https://doi.org/10.1145/3335772.3335936.

[23] Gabriel Bracha and Sam Toueg. “Asynchronous Consensus and Broadcast Protocols”. In: *J. ACM* 32.4 (1985), pp. 824–840. DOI: 10.1145/4221.214134. URL: https://doi.org/10.1145/4221.214134.

[24] Pierre Civit et al. “Crime and Punishment in Distributed Byzantine Decision Tasks”. In: *42nd IEEE International Conference on Distributed Computing Systems, ICDCS 2022, Bologna, Italy, July 10-13, 2022*. IEEE, 2022, pp. 34–44. DOI: 10.1109/ICDCS54860.2022.00013. URL: https://doi.org/10.1109/ICDCS54860.2022.00013.

[25] Christian Cachin et al. “Secure and Efficient Asynchronous Broadcast Protocols”. In: *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*. Ed. by Joe Kilian. Vol. 2139. Lecture Notes in Computer Science. Springer, 2001, pp. 524–541. DOI: 10.1007/3-540-44647-8_31. URL: https://doi.org/10.1007/3-540-44647-8%5C_31.

[26] Benny Chor et al. “Verifiable Secret Sharing and Achieving Simultaneity in the Presence of Faults (Extended Abstract)”. In: *26th Annual Symposium on Foundations of Computer Science, Portland, Oregon, USA, 21-23 October 1985*. IEEE Computer Society, 1985, pp. 383–395. DOI: 10.1109/SFCS.1985.64. URL: https://doi.org/10.1109/SFCS.1985.64.

[27] Kenneth P. Birman and Thomas A. Joseph. “Reliable Communication in the Presence of Failures”. In: *ACM Trans. Comput. Syst.* 5.1 (1987), pp. 47–76. DOI: 10.1145/7351.7478. URL: https://doi.org/10.1145/7351.7478.

[28] Chi Ho, Danny Dolev, and Robbert van Renesse. “Making Distributed Applications Robust”. In: *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*. Ed. by Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal. Vol. 4878. Lecture Notes in Computer Science. Springer, 2007, pp. 232–246. DOI: 10.1007/978-3-540-77096-1_17. URL: https://doi.org/10.1007/978-3-540-77096-1%5C_17.

[29] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system”. In: *Concurrency: the Works of Leslie Lamport*. Ed. by Dahlia Malkhi. ACM, 2019, pp. 179–196. DOI: 10.1145/3335772.3335934. URL: https://doi.org/10.1145/3335772.3335934.

[30] Vassos Hadzilacos and Sam Toueg. “Fault-tolerant broadcasts and related problems”. In: *Distributed systems (2nd Ed.)* 1993, pp. 97–145.

[31] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. “Signature-Free Asynchronous Binary Byzantine Consensus with t $<$ n/3, O(n2) Messages, and O(1) Expected Time”. In: *J. ACM* 62.4 (2015), 31:1–31:21. DOI: 10.1145/2785953. URL: https://doi.org/10.1145/2785953.

[32] Andrew Miller et al. “The Honey Badger of BFT Protocols”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. Ed. by Edgar R. Weippl et al. ACM, 2016, pp. 31–42. DOI: 10.1145/2976749.2978399. URL: https://doi.org/10.1145/2976749.2978399.

[33] Tyler Crain et al. “DBFT: Efficient Leaderless Byzantine Consensus and its Application to Blockchains”. In: *17th IEEE International Symposium on Network Computing and Applications, NCA 2018, Cambridge, MA, USA, November 1-3, 2018*. IEEE, 2018, pp. 1–8. DOI: 10.1109/NCA.2018.8548057. URL: https://doi.org/10.1109/NCA.2018.8548057.

[34] Pierre Civit, Seth Gilbert, and Vincent Gramoli. "Polygraph: Accountable Byzantine Agreement". In: *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*. IEEE, 2021, pp. 403–413. DOI: 10.1109/ICDCS51616.2021.00046. URL: https://doi.org/10.1109/ICDCS51616.2021.00046.

[35] Flaviu Cristian. "Understanding Fault-Tolerant Distributed Systems". In: *Commun. ACM* 34.2 (1991), pp. 56–78. DOI: 10.1145/102792.102801. URL: https://doi.org/10.1145/102792.102801.

[36] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *Proceedings of the Second ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 21-23, 1983, Colony Square Hotel, Atlanta, Georgia, USA*. Ed. by Ronald Fagin and Philip A. Bernstein. ACM, 1983, pp. 1–7. DOI: 10.1145/588058.588060. URL: https://doi.org/10.1145/588058.588060.

[37] James Aspnes and Maurice Herlihy. "Fast Randomized Consensus Using Shared Memory". In: *J. Algorithms* 11.3 (1990), pp. 441–461. DOI: 10.1016/0196-6774(90)90021-6. URL: https://doi.org/10.1016/0196-6774(90)90021-6.

[38] Michael O. Rabin. "Randomized Byzantine Generals". In: *24th Annual Symposium on Foundations of Computer Science, Tucson, Arizona, USA, 7-9 November 1983*. IEEE Computer Society, 1983, pp. 403–409. DOI: 10.1109/SFCS.1983.48. URL: https://doi.org/10.1109/SFCS.1983.48.

[39] Michael Ben-Or. "Fast Asynchronous Byzantine Agreement (Extended Abstract)". In: *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 5-7, 1985*. Ed. by Michael A. Malcolm and H. Raymond Strong. ACM, 1985, pp. 149–151. DOI: 10.1145/323596.323609. URL: https://doi.org/10.1145/323596.323609.

[40] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. "Consensus in the presence of partial synchrony". In: *J. ACM* 35.2 (1988), pp. 288–323. DOI: 10.1145/42282.42283. URL: https://doi.org/10.1145/42282.42283.

[41] Danny Dolev and H. Raymond Strong. "Authenticated Algorithms for Byzantine Agreement". In: *SIAM J. Comput.* 12.4 (1983), pp. 656–666. DOI: 10.1137/0212045. URL: https://doi.org/10.1137/0212045.

[42] Jonathan Katz and Chiu-Yuen Koo. "On Expected Constant-Round Protocols for Byzantine Agreement". In: *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2006, Proceedings*. Ed. by Cynthia Dwork. Vol. 4117. Lecture Notes in Computer Science. Springer, 2006, pp. 445–462. DOI: 10.1007/11818175_27. URL: https://doi.org/10.1007/11818175%5C_27.

[43] Daniel Collins et al. “Online Payments by Merely Broadcasting Messages”. In: *50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2020, Valencia, Spain, June 29 - July 2, 2020*. IEEE, 2020, pp. 26–38. DOI: 10.1109/DSN48063.2020.00023. URL: https://doi.org/10.1109/DSN48063.2020.00023.

[44] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. “Mencius: Building Efficient Replicated State Machine for WANs”. In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 369–384. URL: http://www.usenix.org/events/osdi08/tech/full%5C_papers/mao/mao.pdf.

[45] Iulian Moraru, David G. Andersen, and Michael Kaminsky. “There is more consensus in Egalitarian parliaments”. In: *ACM SIGOPS 24th Symposium on Operating Systems Principles, SOSP ’13, Farmington, PA, USA, November 3-6, 2013*. Ed. by Michael Kaminsky and Mike Dahlin. ACM, 2013, pp. 358–372. DOI: 10.1145/2517349.2517350. URL: https://doi.org/10.1145/2517349.2517350.

[46] Fatemeh Borran and André Schiper. “A Leader-Free Byzantine Consensus Algorithm”. In: *Distributed Computing and Networking, 11th International Conference, ICDCN 2010, Kolkata, India, January 3-6, 2010. Proceedings*. Ed. by Krishna Kant et al. Vol. 5935. Lecture Notes in Computer Science. Springer, 2010, pp. 67–78. DOI: 10.1007/978-3-642-11322-2_11. URL: https://doi.org/10.1007/978-3-642-11322-2%5C_11.

[47] Leslie Lamport. *Leaderless Byzantine consensus*. United States Patent, Microsoft, Redmond, WA (USA). 2010.

[48] Leslie Lamport. “Brief Announcement: Leaderless Byzantine Paxos”. In: *Distributed Computing - 25th International Symposium, DISC 2011, Rome, Italy, September 20-22, 2011. Proceedings*. Ed. by David Peleg. Vol. 6950. Lecture Notes in Computer Science. Springer, 2011, pp. 141–142. DOI: 10.1007/978-3-642-24100-0_10. URL: https://doi.org/10.1007/978-3-642-24100-0%5C_10.

[49] Yuan Lu et al. “Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited”. In: *PODC ’20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*. Ed. by Yuval Emek and Christian Cachin. ACM, 2020, pp. 129–138. DOI: 10.1145/3382734.3405707. URL: https://doi.org/10.1145/3382734.3405707.

[50] Bingyong Guo et al. “Dumbo: Faster Asynchronous BFT Protocols”. In: *CCS ’20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti et al. ACM, 2020, pp. 803–818. DOI: 10.1145/3372297.3417262. URL: https://doi.org/10.1145/3372297.3417262.

[51] Pierre Tholoniat and Vincent Gramoli. “Formal Verification of Blockchain Byzantine Fault Tolerance”. In: *6th Workshop on Formal Reasoning in Distributed Algorithms (FRIDA’19)*. 2019.

[52] Nathalie Bertrand et al. *Compositional Verification of Byzantine Consensus*. Tech. rep. hal-03158911. HAL, Mar. 2021. URL: https://hal.archives-ouvertes.fr/hal-03158911.

[53] Karolos Antoniadis et al. "State Machine Replication Is More Expensive Than Consensus". In: *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*. Ed. by Ulrich Schmid and Josef Widder. Vol. 121. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 7:1–7:18. DOI: 10.4230/LIPICS.DISC.2018.7. URL: https://doi.org/10.4230/LIPIcs.DISC.2018.7.

[54] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Comput. Surv.* 22.4 (1990), pp. 299–319. DOI: 10.1145/98163.98167. URL: https://doi.org/10.1145/98163.98167.

[55] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance". In: *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana, USA, February 22-25, 1999*. Ed. by Margo I. Seltzer and Paul J. Leach. USENIX Association, 1999, pp. 173–186. URL: https://dl.acm.org/citation.cfm?id=296824.

[56] Ramakrishna Kotla et al. "Zyzzyva: Speculative Byzantine fault tolerance". In: *ACM Trans. Comput. Syst.* 27.4 (2009), 7:1–7:39. DOI: 10.1145/1658357.1658358. URL: https://doi.org/10.1145/1658357.1658358.

[57] Allen Clement et al. "Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults". In: *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009, April 22-24, 2009, Boston, MA, USA*. Ed. by Jennifer Rexford and Emin Gün Sirer. USENIX Association, 2009, pp. 153–168. URL: http://www.usenix.org/events/nsdi09/tech/full%5C_papers/clement/clement.pdf.

[58] Shengyun Liu et al. "XFT: Practical Fault Tolerance beyond Crashes". In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. Ed. by Kimberly Keeton and Timothy Roscoe. USENIX Association, 2016, pp. 485–500. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/liu.

[59] Diego Ongaro and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the 2014 USENIX Annual Technical Conference, USENIX ATC 2014, Philadelphia, PA, USA, June 19-20, 2014*. Ed. by Garth Gibson and Nickolai Zeldovich. USENIX Association, 2014, pp. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[60] Guy Golan-Gueta et al. "SBFT: A Scalable and Decentralized Trust Infrastructure". In: *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*. IEEE, 2019, pp. 568–580. DOI: 10.1109/DSN.2019.00063. URL: https://doi.org/10.1109/DSN.2019.00063.

[61] Maofan Yin et al. “HotStuff: BFT Consensus with Linearity and Responsiveness”. In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. Ed. by Peter Robinson and Faith Ellen. ACM, 2019, pp. 347–356. DOI: 10.1145/3293611.3331591. URL: https://doi.org/10.1145/3293611.3331591.

[62] Dirk A Zetzsche, Douglas W Arner, and Ross P Buckley. “Decentralized finance (defi)”. In: *Journal of Financial Regulation* 6 (2020), pp. 172–203.

[63] Juan A. Garay, Aggelos Kiayias, and Nikos Leonardos. “The Bitcoin Backbone Protocol: Analysis and Applications”. In: *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9057. Lecture Notes in Computer Science. Springer, 2015, pp. 281–310. DOI: 10.1007/978-3-662-46803-6_10. URL: https://doi.org/10.1007/978-3-662-46803-6%5C_10.

[64] Rafael Pass, Lior Seeman, and Abhi Shelat. “Analysis of the Blockchain Protocol in Asynchronous Networks”. In: *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10211. Lecture Notes in Computer Science. 2017, pp. 643–673. DOI: 10.1007/978-3-319-56614-6_22. URL: https://doi.org/10.1007/978-3-319-56614-6%5C_22.

[65] Rafael Pass and Elaine Shi. “The Sleepy Model of Consensus”. In: *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part II*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10625. Lecture Notes in Computer Science. Springer, 2017, pp. 380–409. DOI: 10.1007/978-3-319-70697-9_14. URL: https://doi.org/10.1007/978-3-319-70697-9%5C_14.

[66] Natkamon Tovanich, Nicolas Soulié, and Petra Isenberg. “Visual Analytics of Bitcoin Mining Pool Evolution: On the Road Toward Stability?” In: *11th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2021, Paris, France, April 19-21, 2021*. IEEE, 2021, pp. 1–5. DOI: 10.1109/NTMS49979.2021.9432675. URL: https://doi.org/10.1109/NTMS49979.2021.9432675.

[67] Aggelos Kiayias et al. “Ouroboros: A Provably Secure Proof-of-Stake Blockchain Protocol”. In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. Lecture Notes in Computer Science. Springer, 2017, pp. 357–388. DOI: 10.1007/978-3-319-63688-7_12. URL: https://doi.org/10.1007/978-3-319-63688-7%5C_12.

[68] Yossi Gilad et al. “Algorand: Scaling Byzantine Agreements for Cryptocurrencies”. In: *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 51–68. DOI: 10.1145/3132747.3132757. URL: https://doi.org/10.1145/3132747.3132757.

[69] Eleftherios Kokoris-Kogias et al. “OmniLedger: A Secure, Scale-Out, Decentralized Ledger via Sharding”. In: *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. IEEE Computer Society, 2018, pp. 583–598. DOI: 10.1109/SP.2018.000-5. URL: https://doi.org/10.1109/SP.2018.000-5.

[70] Leslie Lamport. “Password authentication with insecure communication”. In: *Commun. ACM* 24.11 (Nov. 1981), pp. 770–772. ISSN: 0001-0782. DOI: 10.1145/358790.358797. URL: https://doi.org/10.1145/358790.358797.

[71] Ethan Buchman. “Tendermint: Byzantine fault tolerance in the age of blockchains”. PhD thesis. University of Guelph, 2016.

[72] Elli Androulaki et al. “Hyperledger fabric: a distributed operating system for permissioned blockchains”. In: *Proceedings of the Thirteenth EuroSys Conference, EuroSys 2018, Porto, Portugal, April 23-26, 2018*. Ed. by Rui Oliveira, Pascal Felber, and Y. Charlie Hu. ACM, 2018, 30:1–30:15. DOI: 10.1145/3190508.3190538. URL: https://doi.org/10.1145/3190508.3190538.

[73] Mathieu Baudet et al. “State machine replication in the libra blockchain”. In: *The Libra Assn., Tech. Rep* 7 (2019).

[74] Tyler Crain, Christopher Natoli, and Vincent Gramoli. “Red Belly: A Secure, Fair and Scalable Open Blockchain”. In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 466–483. DOI: 10.1109/SP40001.2021.00087. URL: https://doi.org/10.1109/SP40001.2021.00087.

[75] Cynthia Dwork and Moni Naor. “Pricing via Processing or Combatting Junk Mail”. In: *Advances in Cryptology - CRYPTO ’92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*. Ed. by Ernest F. Brickell. Vol. 740. Lecture Notes in Computer Science. Springer, 1992, pp. 139–147. DOI: 10.1007/3-540-48071-4_10. URL: https://doi.org/10.1007/3-540-48071-4%5C_10.

[76] XiaoFeng Wang and Michael K. Reiter. “Defending Against Denial-of-Service Attacks with Puzzle Auction”. In: *2003 IEEE Symposium on Security and Privacy (S&P 2003), 11-14 May 2003, Berkeley, CA, USA*. IEEE Computer Society, 2003, pp. 78–92. DOI: 10.1109/SECPRI.2003.1199329. URL: https://doi.org/10.1109/SECPRI.2003.1199329.

[77] Mehran S. Fallah. “A Puzzle-Based Defense Strategy Against Flooding Attacks Using Game Theory”. In: *IEEE Trans. Dependable Secur. Comput.* 7.1 (2010), pp. 5–19. DOI: 10.1109/TDSC.2008.13. URL: https://doi.org/10.1109/TDSC.2008.13.

[78] Yongdong Wu et al. "Software Puzzle: A Countermeasure to Resource-Inflated Denial-of-Service Attacks". In: *IEEE Trans. Inf. Forensics Secur.* 10.1 (2015), pp. 168–177. DOI: 10.1109/TIFS.2014.2366293. URL: https://doi.org/10.1109/TIFS.2014.2366293.

[79] Zhang Laishun, Zhang Minglei, and Guo Yuanbo. "A client puzzle based defense mechanism to resist DoS attacks in WLAN". In: *2010 International Forum on Information Technology and Applications*. Vol. 3. IEEE. 2010, pp. 424–427.

[80] Silvio Micali, Michael O. Rabin, and Salil P. Vadhan. "Verifiable Random Functions". In: *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*. IEEE Computer Society, 1999, pp. 120–130. DOI: 10.1109/SFFCS.1999.814584. URL: https://doi.org/10.1109/SFFCS.1999.814584.

[81] Michael K. Reiter and Kenneth P. Birman. "How to Securely Replicate Services". In: *ACM Trans. Program. Lang. Syst.* 16.3 (1994), pp. 986–1009. DOI: 10.1145/177492.177745. URL: https://doi.org/10.1145/177492.177745.

[82] Sisi Duan, Michael K. Reiter, and Haibin Zhang. "Secure Causal Atomic Broadcast, Revisited". In: *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017*. IEEE Computer Society, 2017, pp. 61–72. DOI: 10.1109/DSN.2017.64. URL: https://doi.org/10.1109/DSN.2017.64.

[83] Michael Lewis. *Flash boys: a Wall Street revolt*. WW Norton & Company, 2014.

[84] *Code of Ethics (Rule 17j-1)*. Accessed: 2022-07-18. 1940. URL: https://www.sec.gov/Archives/edgar/data/1597219/000119312514060646/d667780dex99r1.htm.

[85] Sen Yang et al. "SoK: MEV Countermeasures: Theory and Practice". In: *CoRR* abs/2212.05111 (2022). DOI: 10.48550/ARXIV.2212.05111. arXiv: 2212.05111. URL: https://doi.org/10.48550/arXiv.2212.05111.

[86] Kaihua Qin, Liyi Zhou, and Arthur Gervais. "Quantifying Blockchain Extractable Value: How dark is the forest?" In: *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 2022, pp. 198–214. DOI: 10.1109/SP46214.2022.9833734. URL: https://doi.org/10.1109/SP46214.2022.9833734.

[87] *MEV-explore*. Accessed: 2022-09-12. 2022. URL: https://explore.flashbots.net/.

[88] Liyi Zhou et al. "On the Just-In-Time Discovery of Profit-Generating Transactions in DeFi Protocols". In: *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 919–936. DOI: 10.1109/SP40001.2021.00113. URL: https://doi.org/10.1109/SP40001.2021.00113.

[89] Kushal Babel et al. "Clockwork Finance: Automated Analysis of Economic Security in Smart Contracts". In: *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 2499–2516. DOI: 10.1109/SP46215.2023.10179346. URL: https://doi.org/10.1109/SP46215.2023.10179346.

[90] Lioba Heimbach, Eric Schertenleib, and Roger Wattenhofer. "The Potential of Self-Regulation for Front-Running Prevention on DEXes". In: *CoRR* abs/2306.05756 (2023). DOI: 10.48550/ARXIV.2306.05756. arXiv: 2306.05756. URL: https://doi.org/10.48550/arXiv.2306.05756.

[91] John F Nash et al. "Non-cooperative games". In: (1950).

[92] Colin J Fidge. "Timestamps in message-passing systems that preserve the partial ordering". In: (1987).

[93] Friedemann Mattern et al. *Virtual time and global states of distributed systems*. Univ., Department of Computer Science, 1988.

[94] Matthias Fitzi and Juan A. Garay. "Efficient player-optimal protocols for strong and differential consensus". In: *Proceedings of the Twenty-Second ACM Symposium on Principles of Distributed Computing, PODC 2003, Boston, Massachusetts, USA, July 13-16, 2003*. Ed. by Elizabeth Borowsky and Sergio Rajsbaum. ACM, 2003, pp. 211–220. DOI: 10.1145/872035.872066. URL: https://doi.org/10.1145/872035.872066.

[95] Ke Mu et al. "Separation is Good: A Faster Order-Fairness Byzantine Consensus". In: *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024. URL: https://www.ndss-symposium.org/ndss-paper/separation-is-good-a-faster-order-fairness-byzantine-consensus/.

[96] Klaus Kursawe. "Wendy, the Good Little Fairness Widget: Achieving Order Fairness for Blockchains". In: *AFT '20: 2nd ACM Conference on Advances in Financial Technologies, New York, NY, USA, October 21-23, 2020*. ACM, 2020, pp. 25–36. DOI: 10.1145/3419614.3423263. URL: https://doi.org/10.1145/3419614.3423263.

[97] Florian Suri-Payer et al. "Basil: Breaking up BFT with ACID (transactions)". In: *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*. Ed. by Robbert van Renesse and Nickolai Zeldovich. ACM, 2021, pp. 1–17. DOI: 10.1145/3477132.3483552. URL: https://doi.org/10.1145/3477132.3483552.

[98] Klaus Kursawe. "Wendy Grows Up: More Order Fairness". In: *Financial Cryptography and Data Security. FC 2021 International Workshops - CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers*. Ed. by Matthew Bernhard et al. Vol. 12676. Lecture Notes in Computer Science. Springer, 2021, pp. 191–196. DOI: 10.1007/978-3-662-63958-0_17. URL: https://doi.org/10.1007/978-3-662-63958-0%5C_17.

[99] Lioba Heimbach and Roger Wattenhofer. "SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance". In: *Proceedings of the 4th ACM Conference on Advances in Financial Technologies, AFT 2022, Cambridge, MA, USA, September 19-21, 2022*. Ed. by Maurice Herlihy and Neha Narula. ACM, 2022, pp. 47–60. DOI: 10.1145/3558535.3559784. URL: https://doi.org/10.1145/3558535.3559784.

[100] Yuan Lu et al. "Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited". In: *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020*. Ed. by Yuval Emek and Christian Cachin. ACM, 2020, pp. 129–138. DOI: 10.1145/3382734.3405707. URL: https://doi.org/10.1145/3382734.3405707.

[101] Bingyong Guo et al. "Dumbo: Faster Asynchronous BFT Protocols". In: *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. Ed. by Jay Ligatti et al. ACM, 2020, pp. 803–818. DOI: 10.1145/3372297.3417262. URL: https://doi.org/10.1145/3372297.3417262.

[102] Avi Asayag et al. "A Fair Consensus Protocol for Transaction Ordering". In: *2018 IEEE 26th International Conference on Network Protocols, ICNP 2018, Cambridge, UK, September 25-27, 2018*. IEEE Computer Society, 2018, pp. 55–65. DOI: 10.1109/ICNP.2018.00016. URL: https://doi.org/10.1109/ICNP.2018.00016.

[103] Adi Shamir. "How to Share a Secret". In: *Commun. ACM* 22.11 (1979), pp. 612–613. DOI: 10.1145/359168.359176. URL: https://doi.org/10.1145/359168.359176.

[104] Chrysoula Stathakopoulou et al. "Adding Fairness to Order: Preventing Front-Running Attacks in BFT Protocols using TEEs". In: *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021*. IEEE, 2021, pp. 34–45. DOI: 10.1109/SRDS53918.2021.00013. URL: https://doi.org/10.1109/SRDS53918.2021.00013.

[105] Suyash Gupta et al. "Dissecting BFT Consensus: In Trusted Components we Trust!" In: *Proceedings of the Eighteenth European Conference on Computer Systems, EuroSys 2023, Rome, Italy, May 8-12, 2023*. Ed. by Giuseppe Antonio Di Luna et al. ACM, 2023, pp. 521–539. DOI: 10.1145/3552326.3587455. URL: https://doi.org/10.1145/3552326.3587455.

[106] Jian Liu et al. "Scalable Byzantine Consensus via Hardware-Assisted Secret Sharing". In: *IEEE Trans. Computers* 68.1 (2019), pp. 139–151. DOI: 10.1109/TC.2018.2860009. URL: https://doi.org/10.1109/TC.2018.2860009.

[107] Jiashuo Zhang et al. "TBFT: Efficient Byzantine Fault Tolerance Using Trusted Execution Environment". In: *IEEE International Conference on Communications, ICC 2022, Seoul, Korea, May 16-20, 2022*. IEEE, 2022, pp. 1004–1009. DOI: 10.1109/ICC45855.2022.9839169. URL: https://doi.org/10.1109/ICC45855.2022.9839169.

[108] Joshua Lind et al. "Teechain: a secure payment network with asynchronous blockchain access". In: *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*. Ed. by Tim Brecht and Carey Williamson. ACM, 2019, pp. 63–79. DOI: 10.1145/3341301.3359627. URL: https://doi.org/10.1145/3341301.3359627.

[109] Jinghui Liao et al. “Speedster: An Efficient Multi-party State Channel via Enclaves”. In: *ASIA CCS '22: ACM Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May 2022 - 3 June 2022*. Ed. by Yuji Suga et al. ACM, 2022, pp. 637–651. DOI: 10.1145/3488932.3523259. URL: https://doi.org/10.1145/3488932.3523259.

[110] Whitfield Diffie and Martin E. Hellman. “New Directions in Cryptography”. In: *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. Ed. by Rebecca Slayton. Vol. 42. ACM Books. ACM, 2022, pp. 365–390. DOI: 10.1145/3549993.3550007. URL: https://doi.org/10.1145/3549993.3550007.

[111] Yvo Desmedt. “Society and Group Oriented Cryptography: A New Concept”. In: *Advances in Cryptology - CRYPTO '87, A Conference on the Theory and Applications of Cryptographic Techniques, Santa Barbara, California, USA, August 16-20, 1987, Proceedings*. Ed. by Carl Pomerance. Vol. 293. Lecture Notes in Computer Science. Springer, 1987, pp. 120–127. DOI: 10.1007/3-540-48184-2_8. URL: https://doi.org/10.1007/3-540-48184-2%5C_8.

[112] Yvo Desmedt and Yair Frankel. “Threshold Cryptosystems”. In: *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*. Ed. by Gilles Brassard. Vol. 435. Lecture Notes in Computer Science. Springer, 1989, pp. 307–315. DOI: 10.1007/0-387-34805-0_28. URL: https://doi.org/10.1007/0-387-34805-0%5C_28.

[113] Gabriel Bracha. “Asynchronous Byzantine Agreement Protocols”. In: *Inf. Comput.* 75.2 (1987), pp. 130–143. DOI: 10.1016/0890-5401(87)90054-X. URL: https://doi.org/10.1016/0890-5401(87)90054-X.

[114] Xavier Défago, André Schiper, and Péter Urbán. “Total order broadcast and multicast algorithms: Taxonomy and survey”. In: *ACM Comput. Surv.* 36.4 (2004), pp. 372–421. DOI: 10.1145/1041680.1041682. URL: https://doi.org/10.1145/1041680.1041682.

[115] Achour Mostéfaoui and Michel Raynal. “Intrusion-Tolerant Broadcast and Agreement Abstractions in the Presence of Byzantine Processes”. In: *IEEE Trans. Parallel Distributed Syst.* 27.4 (2016), pp. 1085–1098. DOI: 10.1109/TPDS.2015.2427797. URL: https://doi.org/10.1109/TPDS.2015.2427797.

[116] Idit Keidar et al. “All You Need is DAG”. In: *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*. Ed. by Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen. ACM, 2021, pp. 165–175. DOI: 10.1145/3465084.3467905. URL: https://doi.org/10.1145/3465084.3467905.

[117] George Danezis et al. “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*. Ed. by Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis. ACM, 2022, pp. 34–50. DOI: 10.1145/3492321.3519594. URL: https://doi.org/10.1145/3492321.3519594.

[118] Dahlia Malkhi and Michael K. Reiter. "Byzantine Quorum Systems". In: *Distributed Comput.* 11.4 (1998), pp. 203–213. DOI: 10.1007/S004460050050. URL: https://doi.org/10.1007/s004460050050.

[119] Nathalie Bertrand et al. "Brief Announcement: Holistic Verification of Blockchain Consensus". In: *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*. Ed. by Alessia Milani and Philipp Woelfel. ACM, 2022, pp. 424–426. DOI: 10.1145/3519270.3538468. URL: https://doi.org/10.1145/3519270.3538468.

[120] Nathalie Bertrand et al. "Holistic Verification of Blockchain Consensus". In: *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*. Ed. by Christian Scheideler. Vol. 246. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 10:1–10:24. DOI: 10.4230/LIPICS.DISC.2022.10. URL: https://doi.org/10.4230/LIPIcs.DISC.2022.10.

[121] Shai Halevi and Silvio Micali. "Practical and Provably-Secure Commitment Schemes from Collision-Free Hashing". In: *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, 1996, pp. 201–215. DOI: 10.1007/3-540-68697-5_16. URL: https://doi.org/10.1007/3-540-68697-5%5C_16.

[122] Maxime Mouchet, Sandrine Vaton, and Thierry Chonavel. "Statistical Characterization of Round-Trip Times with Nonparametric Hidden Markov Models". In: *IFIP/IEEE International Symposium on Integrated Network Management, IM 2019, Washington, DC, USA, April 09-11, 2019*. Ed. by Joe Betser et al. IFIP, 2019, pp. 43–48. URL: https://ieeexplore.ieee.org/document/8717870.

[123] Joseph L Doob. "What is a Martingale?" In: *The American Mathematical Monthly* 78.5 (1971), pp. 451–463.

[124] Lioba Heimbach and Roger Wattenhofer. "SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance". In: *Proceedings of the 4th ACM Conference on Advances in Financial Technologies, AFT 2022, Cambridge, MA, USA, September 19-21, 2022*. Ed. by Maurice Herlihy and Neha Narula. ACM, 2022, pp. 47–60. DOI: 10.1145/3558535.3559784. URL: https://doi.org/10.1145/3558535.3559784.

[125] Mary Maller et al. "Sonic: Zero-Knowledge SNARKs from Linear-Size Universal and Updatable Structured Reference Strings". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. Ed. by Lorenzo Cavallaro et al. ACM, 2019, pp. 2111–2128. DOI: 10.1145/3319535.3339817. URL: https://doi.org/10.1145/3319535.3339817.

[126] Sourav Das, Zhuolun Xiang, and Ling Ren. "Asynchronous Data Dissemination and its Applications". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*. Ed. by

Yongdae Kim et al. ACM, 2021, pp. 2705–2721. DOI: 10.1145/3460120.3484808. URL: https://doi.org/10.1145/3460120.3484808.

[127] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. "Asymptotically Optimal Validated Asynchronous Byzantine Agreement". In: *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*. Ed. by Peter Robinson and Faith Ellen. ACM, 2019, pp. 337–346. DOI: 10.1145/3293611.3331612. URL: https://doi.org/10.1145/3293611.3331612.

[128] Fatima M. Anwar et al. "Securing Time in Untrusted Operating Systems with TimeSeal". In: *IEEE Real-Time Systems Symposium, RTSS 2019, Hong Kong, SAR, China, December 3-6, 2019*. IEEE, 2019, pp. 80–92. DOI: 10.1109/RTSS46320.2019.00018. URL: https://doi.org/10.1109/RTSS46320.2019.00018.

[129] Marco Chiesa et al. "A Survey of Fast-Recovery Mechanisms in Packet-Switched Networks". In: *IEEE Commun. Surv. Tutorials* 23.2 (2021), pp. 1253–1301. DOI: 10.1109/COMST.2021.3063980. URL: https://doi.org/10.1109/COMST.2021.3063980.

[130] Christopher Natoli and Vincent Gramoli. "The Blockchain Anomaly". In: *15th IEEE International Symposium on Network Computing and Applications, NCA 2016, Cambridge, Boston, MA, USA, October 31 - November 2, 2016*. Ed. by Alessandro Pellegrini et al. IEEE Computer Society, 2016, pp. 310–317. DOI: 10.1109/NCA.2016.7778635. URL: https://doi.org/10.1109/NCA.2016.7778635.

[131] Maxime Mouchet et al. "Large-Scale Characterization and Segmentation of Internet Path Delays With Infinite HMMs". In: *IEEE Access* 8 (2020), pp. 16771–16784. DOI: 10.1109/ACCESS.2020.2968380. URL: https://doi.org/10.1109/ACCESS.2020.2968380.

[132] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not". In: *2015 IEEE TrustCom/BigDataSE/ISPA, Helsinki, Finland, August 20-22, 2015, Volume 1*. IEEE, 2015, pp. 57–64. DOI: 10.1109/TRUSTCOM.2015.357. URL: https://doi.org/10.1109/Trustcom.2015.357.

[133] Frank McKeen et al. "Innovative instructions and software model for isolated execution". In: *HASP 2013, The Second Workshop on Hardware and Architectural Support for Security and Privacy, Tel-Aviv, Israel, June 23-24, 2013*. Ed. by Ruby B. Lee and Weidong Shi. ACM, 2013, p. 10. DOI: 10.1145/2487726.2488368. URL: https://doi.org/10.1145/2487726.2488368.

[134] Fatima M. Anwar and Mani B. Srivastava. "Applications and Challenges in Securing Time". In: *12th USENIX Workshop on Cyber Security Experimentation and Test, CSET 2019, Santa Clara, CA, USA, August 12, 2019*. Ed. by Rob Jansen and Peter A. H. Peterson. USENIX Association, 2019. URL: https://www.usenix.org/conference/cset19/presentation/anwar.

[135] Jennifer Lundelius and Nancy A. Lynch. "A New Fault-Tolerant Algorithm for Clock Synchronization". In: *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27-29, 1984*. Ed. by Tiko Kameda et al. ACM, 1984, pp. 75–88. DOI: 10.1145/800222.806738. URL: https://doi.org/10.1145/800222.806738.

[136] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. "Optimal clock synchronization in networks". In: *Proceedings of the 7th International Conference on Embedded Networked Sensor Systems, SenSys 2009, Berkeley, California, USA, November 4-6, 2009*. Ed. by David E. Culler, Jie Liu, and Matt Welsh. ACM, 2009, pp. 225–238. DOI: 10.1145/1644038.1644061. URL: https://doi.org/10.1145/1644038.1644061.

[137] Karolos Antoniadis et al. "Leaderless Consensus". In: *41st IEEE International Conference on Distributed Computing Systems, ICDCS 2021, Washington DC, USA, July 7-10, 2021*. IEEE, 2021, pp. 392–402. DOI: 10.1109/ICDCS51616.2021.00045. URL: https://doi.org/10.1109/ICDCS51616.2021.00045.

[138] Yahui Zhang et al. "Survey of attacks and defenses against SGX". In: *2020 IEEE 5th Information Technology and Mechatronics Engineering Conference (ITOEC)*. IEEE. 2020, pp. 1492–1496.

[139] Kit Murdock et al. "Plundervolt: Software-based Fault Injection Attacks against Intel SGX". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1466–1482. DOI: 10.1109/SP40000.2020.00057. URL: https://doi.org/10.1109/SP40000.2020.00057.

[140] Mahimna Kelkar, Soubhik Deb, and Sreeram Kannan. "Order-Fair Consensus in the Permissionless Setting". In: *APKC '22: Proceedings of the 9th ACM on ASIA Public-Key Cryptography Workshop, APKC@AsiaCCS 2022, Nagasaki, Japan, 30 May 2022*. Ed. by Jason Paul Cruz and Naoto Yanai. ACM, 2022, pp. 3–14. DOI: 10.1145/3494105.3526239. URL: https://doi.org/10.1145/3494105.3526239.