Distributed Services Under Attack

Shir Cohen

Distributed Services Under Attack

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Shir Cohen

Submitted to the Senate
of the Technion — Israel Institute of Technology
Tammuz 5783 Haifa July 2023

This research was carried out under the supervision of Prof. Idit Keidar, in the Faculty of Computer Science.

The author of this thesis states that the research, including the collection, processing and presentation of data, addressing and comparing to previous research, etc., was done entirely in an honest way, as expected from scientific research that is conducted according to the ethical standards of the academic world. Also, reporting the research and its results in this thesis was done in an honest and complete manner, according to the same standards.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period, the most up-to-date versions of which being:

Keren Censor-Hillel, Shir Cohen, Ran Gelles, and Gal Sela. Distributed computations in fully-defective networks. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 141–150. ACM, 2022.

Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: sub-quadratic asynchronous byzantine agreement WHP. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, 25:1–25:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

Shir Cohen and Idit Keidar. Tame the wild with byzantine linearizability: reliable broadcast, snapshots, and asset transfer. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, 18:1–18:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

Shir Cohen, Idit Keidar, and Oded Naor. Byzantine agreement with less communication: recent advances. *SIGACT News*, 52(1):71–80, 2021.

Guy Goren, Lefteris Kokoris-Kogias, Alberto Sonnino, Shir Cohen, and Alexander Spiegelman. Proof of availability & retrieval in a modular blockchain architecture. In *Financial Cryptography and Data Security - 27th International Conference, 2023*, 2023.

Shir Cohen, Rati Gelashvili, Eleftherios Kokoris-Kogias, Zekun Li, Dahlia Malkhi, Alberto Sonnino, and Alexander Spiegelman. Be aware of your leaders. In Ittay Eyal and Juan A. Garay, editors, *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*, volume 13411 of *Lecture Notes in Computer Science*, pages 279–295. Springer, 2022.

Shir Cohen, Idit Keidar, and Alexander Spiegelman. Make every word count: adaptive byzantine agreement with fewer words. In *26th International Conference on Principles of Distributed Systems (OPODIS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

Konstantinos Chalkias, Shir Cohen, Kevin Lewi, Fredric Moezinia, and Yolan Romailler. Hashwires: hyperefficient credential-based range proofs. *Privacy Enhancing Technologies Symposium (PETS 2021)*, 2021.

Acknowledgements

I would like to express my heartfelt gratitude to my advisor, Idit Keidar, for her unwavering guidance, invaluable insights, and endless support throughout this research.

I am grateful to my dedicated research group and friends for productive and enjoyable collaborations during my research. You have been pivotal on this journey.

I would like to share my heartfelt appreciation to my family for their continuous encouragement and motivation. Your belief in my abilities pushed me to strive for excellence.

Finally, and most importantly, I wish to thank my husband, Barak. Your support throughout my doctorate is what made it possible. Your steadfast belief in me was my driving force. Thank you for helping me achieve a ”state of the art” kind of family, together with our amazing daughter Romy. A special shoutout to my loyal canine companion, Zeus, for keeping me company during long hours of work.

“Last but not least, I wanna thank me
I wanna thank me for believing in me
I wanna thank me for doing all this hard work
I wanna thank me for having no days off
I wanna thank me for, for never quitting.”
S. Dogg

.

The generous financial help of the Technion, Hasso Plattner Institute Scholarship, and the Israel Academy Adams Fellowship is gratefully acknowledged.

Contents

List of Figures

Abstract

This thesis focuses on distributed systems and their important properties such as fault tolerance and scalability. The goal is to examine services implemented in distributed systems that are prone to arbitrary Byzantine faults. The use of distributed systems, including blockchains, has increased in recent years, posing new challenges in terms of reliability and scalability.

The thesis begins by discussing three key services in distributed computing: State Machine Replication (SMR), Byzantine Agreement (BA), and Cryptocurrencies. State Machine Replication is a fault-tolerant service composed of replicated servers that act as state machines, ensuring that correct replicas follow the same sequence of state transitions triggered by client requests. Byzantine Agreement is a problem where a set of correct processes aim to reach a common decision despite the presence of malicious Byzantine processes. Cryptocurrencies allow for distributed management of clients' assets, and a State Machine Replication can be used to implement them.

The thesis then delves into the background of Byzantine Agreement, discussing different timing models (synchronous, eventual synchrony, and fully asynchronous) and the complexity lower bounds associated with them. It highlights the recent advancements in sub-quadratic Byzantine Agreement algorithms using randomness and cryptographic tools to achieve improved communication complexity.

The results presented in the thesis include the introduction of a sub-quadratic asynchronous Byzantine Agreement algorithm that surpasses existing solutions in terms of scalability and a synchronous Byzantine Agreement algorithm with adaptive complexity. The thesis also proposes a leader-rotation mechanism called Carousel that ensures positive Chain-quality and limits the number of faulty leaders in crash-only executions. Additionally, it explores optimal-resilience algorithms for concurrent shared-memory objects like asset transfer, reliable broadcast, and snapshot, demonstrating their implementation from registers.

Overall, this thesis contributes to the field of distributed systems by addressing key challenges and providing solutions for fault tolerance, scalability, and reliability in the context of various timing models and cryptographic techniques.

Notation and Abbreviations

BA	Byzantine Agreement
BB	Byzantine Broadcast
ES	Eventual Synchrony
GST	global stabilization time
LBR	Leader-based round
MWMR	Multi-Writer-Multi-Reader (Register)
PKI	public key infrastructure
SMR	State Machine Replication
SWMR	Single-Writer-Multi-Reader (Register)
VRF	verifiable random functions
WHP	with high probability

Chapter 1

Introduction

A distributed system is a computer system in which multiple independent processes work together to achieve a common goal. These processes can be located in different geographical locations and communicate with each other to convey information upon which they act. Such systems are used daily by billions of users, for example by accessing email services or cloud storage. The main advantage of a distributed system is that it can provide increased scalability, fault tolerance, and reliability compared to a centralized system.

Fault tolerance is an important property in distributed systems. It represents the ability of the system to stay correct and make progress despite the failures of some subsets of the processes. Therefore, it is important to model these failures in theoretical settings and to construct algorithms that provide this property.

Different models capture different failures of the processes. Some handle crash failures, where processes may stop responding, and some handle Byzantine failures. The latter describes processes that may deviate arbitrarily from the protocol. In particular, they may crash, fail to send or receive messages, and send arbitrary messages. In both cases, it is common to model failures by assuming an adversary that determines the failure pattern and to construct algorithms that are resilient to the worst-case pattern.

Another important property is scalability. Scalability captures the ability to add resources to the system (e.g., increase the number of processes) in order to improve the system's performance and/or to support a bigger number of users. In simpler terms, it refers to a system's ability to grow or expand in size or complexity to meet changing demands.

In this thesis, we seek to examine services that are implemented in distributed systems and that are prone to arbitrary faults. The use of distributed systems has increased in the last decades, with a new use case of blockchains – decentralized and distributed digital ledgers that record transactions across a network of computers. As a growing crowd uses these systems, guaranteeing properties such as reliability and scalability faces new challenges. Since large and growing deployed real-world systems are distributed, it is important to study different algorithms and primitives of distributed computing that provide the above-mentioned properties. In this introduction chapter, we start by presenting the services discussed in this thesis (Section 1.1). Next, in Section 1.2 we cover the background that is the starting point of our work. Finally, in Section 1.3 we briefly go over the results presented in this thesis.

1.1 State Machine Replication, Byzantine Agreement, and Cryptocurrency

Some of the most useful primitives in distributed computing are *State Machine Replication* (SMR), *Byzantine Agreement* (BA), and *Cryptocurrencies*. SMR is a primitive that provides a fault-tolerant service used by clients. It is composed of replicated servers that operate in a deterministic manner by acting as state machines. It is done by ensuring that correct replicas follow the same sequence of state transitions. The state transitions are triggered by client requests that are fed into the system. In the context of blockchains, one usually refers to an SMR protocol as the task of a set of processes aiming to maintain a growing chain of *blocks*. Processes participate in a sequence of rounds, attempting to form a block per round. A block contains a set of clients' transactions and some additional metadata, as well as some information that links the block to the previous blocks (see Figure 1.1).

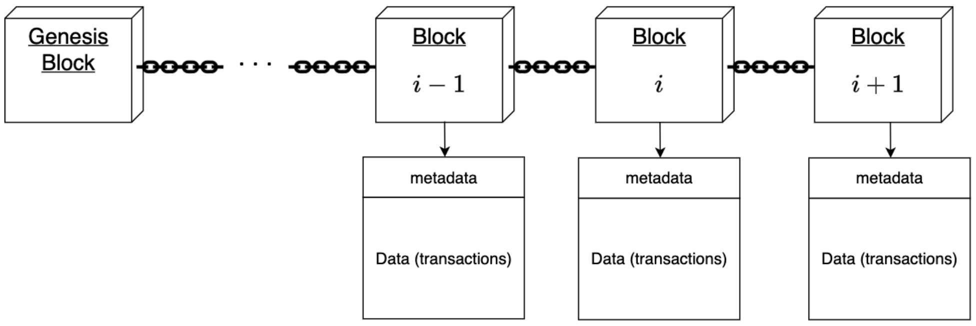

Figure 1.1: Blockchain Scheme.

A Byzantine Agreement primitive (BA) [LSP82] often serves as the main building block for constructing an SMR service. In this problem, a set of correct processes aim to reach a common decision, despite the presence of up to t malicious (Byzantine) ones. In the absence of Byzantine processes, this problem is also known as the *consensus* problem. Using BA solutions, one can implement SMR by deciding upon the next state transition (e.g., the next block) at each point in time.

Finally, another widely-use primitive is asset transfer (also called cryptocurrency). It is used in many applications [Nak09; GHM$^+$17] and allows for distributed management of clients' assets. In its most basic form, it provides each account holder the ability to transfer assets to other accounts and read one's balance. One way to implement an asset transfer is by using SMR, where the transactions are the form of transferring the assets, and the state of the state reflects the balances of different users. That is, a blockchain is a specific implementation of an asset transfer. But as we elaborate in this work, it is not the only form.

The above-mentioned primitives are the starting point of the research that appears in this thesis. Specifically, BA plays an important role in all of the papers included, and Section 1.2 covers the prior work related to it. This section also provides a quick overview of different models under which this primitive, along with other distributed services, are being examined.

Tracing back to the beginning of this introduction, distributed systems usually embody some sort of solution to the consensus problem. As we mentioned, distributed systems are not new and in fact, BA has been studied for four decades now. However, until recently, it has been considered at a fairly small scale. The practical use cases of BA in large-scale systems motivate a push for reduced communication complexity. This goal has guided us in our research work.

1.2 Byzantine Agreement: Background

As a starting point, we begin by stating the notation used in this work. We notate the number of processes in a system by n and use the letter t to denote the threshold of failures in this system. That is, up to t out of n processes may fail according to the discussed adversarial model (crash-fault, Byzantine, etc.). The highest threshold of faults that can be met is called *optimal resilience* and it varies in different models. For example, in a synchronous model with the optimal resilience is $n \geq 2t + 1$ [DS83].

The efficiency of a protocol can be measured by its *word complexity*. The word complexity of a deterministic BA protocol is defined as the number of words all correct processes send until a decision is reached, where a word is a constant number of bits (e.g., the size of a PKI signature). It has been shown by Dolev and Reischuk [DR85] that in deterministic algorithms $\Omega(n^2)$ word complexity is needed in the worst-case, assuming $t = O(n)$. Nevertheless, almost all deterministic works incur a word complexity of at least $O(n^3)$ in a synchronous model with the optimal resilience [DS83; AMN+20]. In fact, this gap remained open for 35 years until recently Momose and Ren [MR20] solved synchronous BA with optimal resilience with $O(n^2)$ words. It is worth mentioning that less resilient solutions with $O(n^2)$ complexity have been known prior to this result.

Yet synchronous solutions are not robust in large-scale systems, where messages can be delayed for extensive periods. A more practical approach is to consider the *eventual synchrony* (ES) model, where communication is initially asynchronous but eventually becomes synchronous. Eventually synchronous algorithms always ensure safety, but their liveness is conditioned on communication becoming timely. In this model, performance is measured during the synchronous period, and the optimal resilience is $n = 3t + 1$. Recent works have used threshold cryptography in order to achieve quadratic complexity in certain optimistic scenarios [AGM18; BCC$^+$19; YMR$^+$19; GLT$^+$20; MR20].

Because attackers may cause communication delays, an even more robust approach is to consider a fully asynchronous model. But in this model, BA cannot be solved deterministically [FLP85]. Although the complexity lower bound does not apply to randomized algorithms, until fairly recently, randomized solutions also required (expected) $O(n^2)$ word complexity [Rab83; CKS05; MMR15; AMS19].

A few recent studies have used randomness to circumvent Dolev and Reischuk's lower bound and provide BA solutions with sub-quadratic word complexity [KS11; GHM$^+$17; Nak09; CKS20; AGM18; BCC$^+$19; Kwo14; NBMS20; Spi21] in all three timing models. In general, there are two approaches to using probability in achieving this goal. The first is weakening the problem guarantees to probabilistic ones. Works in this vein usually utilize committee sampling, assume an adaptive adversary, and provide probabilistic safety and liveness [KS11; GHM$^+$17; Nak09; CKS20]. The second considers models in which deterministic BA solutions are possible, and designs protocols where the *expected* complexity is sub-quadratic [AGM18; BCC$^+$19; Kwo14; NBMS20; Spi21]. The latter works are resilient only to a static adversary whereas the former tolerate a dynamic one. That is, they are resilient to an adversary that has to decide the corruption pattern before the execution of the protocol.

1.3 Results

The first work in this thesis, "Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP" (Chapter 2.1), introduces the first sub-quadratic BA algorithm for an asynchronous message-passing environment. This is a significant improvement over Algorand [GHM$^+$17], one of the leading blockchain companies nowadays. While Algorand is restricted to models with timing assumptions, our solution is completely free of these limitations. This work takes advantage of cryptographic tools (*verifiable random functions* [MRV99]) and uses subtle probability techniques to show that with high probability, a sub-quadratic solution for the BA problem is obtained.

In a later work, "Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words" (Chapter 2.4) we focus on a synchronous model and try to redefine what can be done under such a model. In this work, we present the first BA algorithm with $O(n(f + 1))$ communication complexity and resilience $n = 2t + 1$, where t is an upper bound on process failures in a run and f is the actual number of process failures. We call the communication complexity that depends on f rather than on t an *adaptive complexity.* To achieve this property in this work we take advantage of another cryptographic tool, called *threshold signatures.* This tool allows to aggregate multiple signatures into one-word messages and is commonly used with a threshold of $n-t$. I.e., the number of guaranteed correct processes in the system. Unfortunately, in a system with resilience $n = 2t + 1$, there is not much that can be done with this threshold. Instead, we utilized a threshold on the number of signatures such that on one hand, this number is sufficient to ensure a safe algorithm with adaptive communication in case there are not "many" Byzantine processes. On the other hand, failing to achieve this threshold indicates a high number of failures, which allows the use of a quadratic fallback algorithm.

As discussed BA solutions are commonly used to construct SMRs. And while any BA provides the mechanism to elect the next state in the SMR, most deployed systems in the era of blockchains focus on *leader-based* solutions. In such systems, the series of decisions has context within their position in the sequence. Every decision is driven by a designated leader, which is usually rotated in each round. Leader-rotation is specifically important in a Byzantine setting since processes should not trust each other for load sharing, reward management, resisting censoring of submitted transactions, or ordering requests fairly.

Most existing Leader-rotation mechanisms are implemented in the eventually-synchronous model and use a round-robin approach to rotate leaders [Tea; YMR+19; CS20]. This guarantees that correct processes get a chance to be leaders infinitely often, which is sufficient to drive progress and satisfy a property called Chain-quality [GKL15]. Roughly speaking, the latter stipulates that the number of blocks committed to the chain by correct processes is proportional to the correct processes' percentage. The drawback of such a mechanism is that it does not bound the number of faulty processes which are designated as leaders during an execution. In the paper "Be Aware of Our Leaders" that appears in Chapter 2.3, we propose a leader-rotation mechanism, Carousel, that enjoys both worlds. Carousel satisfies non-zero Chain-quality, and at the same time, bounds the number of faulty leaders in crash-only executions after the global stabilization time (GST), a property we call *Leader-utilization.*

Finally, the asset-transfer primitive is discussed in Chapter 2.2, in the work "Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer". The emerging interest in BA and SMR solutions is their significant role in the construction of blockchains nowadays. However, blockchain is just one technology that solves the asset transfer problem. To find new solutions, we revisit the problem in a shared-memory model. This model differs from the message-passing model discussed so far, in the way that processes communicate among themselves. While the previous works assumed that communication is done via messages they send and receive to one another, we now consider that processes have common registers from which they can read, and can write (different registers have different read-write permissions). In this context, we have undertaken a new research direction. We defined a general correctness notion for concurrent shared-memory objects used by Byzantine processes and systematically studied shared objects in this regard.

In the presented work, we designed optimal-resilience algorithms for some of the most useful concurrent shared-memory objects: asset transfer, reliable broadcast, and snapshot. Reliable broadcast and snapshot are used to implement the asset transfer object but are also of independent interest. We prove that there is an t-resilient implementation of such objects from registers with n processes if and only if $t < \frac{n}{2}$.

Chapter 2

Results

2.1 Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP

Appears in the 34th International Symposium on Distributed Computing (DISC 2020).

Not a COINcidence: Sub-Quadratic Asynchronous Byzantine Agreement WHP

Shir Cohen
Technion, Israel
shirco@campus.technion.ac.il

Idit Keidar
Technion, Israel
idish@ee.technion.ac.il

Alexander Spiegelman
VMware Research, Israel
sasha.spiegelman@gmail.com

Abstract

King and Saia were the first to break the quadratic word complexity bound for Byzantine Agreement in synchronous systems against an adaptive adversary, and Algorand broke this bound with near-optimal resilience (first in the synchronous model and then with eventual-synchrony). Yet the question of asynchronous sub-quadratic Byzantine Agreement remained open. To the best of our knowledge, we are the first to answer this question in the affirmative. A key component of our solution is a shared coin algorithm based on a VRF. A second essential ingredient is VRF-based committee sampling, which we formalize and utilize in the asynchronous model for the first time. Our algorithms work against a delayed-adaptive adversary, which cannot perform after-the-fact removals but has full control of Byzantine processes and full information about communication in earlier rounds. Using committee sampling and our shared coin, we solve Byzantine Agreement with high probability, with a word complexity of $\widetilde{O}(n)$ and $O(1)$ expected time, breaking the $O(n^2)$ bit barrier for asynchronous Byzantine Agreement.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Theory of computation → Cryptographic primitives; Mathematics of computing → Probabilistic algorithms

Keywords and phrases shared coin, Byzantine Agreement, VRF, sub-quadratic consensus protocol

1 Introduction

Byzantine Agreement (*BA*) [27] has been studied for four decades by now, but until recently, has been considered at a fairly small scale. In recent years, however, we begin to see practical use-cases of BA in large-scale systems, which motivates a push for reduced communication complexity. In deterministic algorithms, Dolev and Reischuk's renown lower bound stipulates that $\Omega(n^2)$ communication is needed [17], and until fairly recently, almost all randomized solutions have also had (expected) quadratic word complexity. Recent work has broken this barrier [22, 20, 31], but not in asynchronous settings. We present here the first sub-quadratic asynchronous Byzantine Agreement algorithm. Our algorithm is randomized and solves binary BA *with high probability (whp)*, i.e., with probability that tends to 1 as n goes to infinity.

We consider a system with a static set of n processes, in the so-called "permissioned" setting, where the ids of all processes are well-known. Our algorithm tolerates f failures for $n \approx 4.5f$ (asymptotically). In addition, we assume a trusted *public key infrastructure* (PKI) that allows us to use *verifiable random functions* (VRFs) [29].

We assume a strong adversary that can adaptively take over processes, whereupon it has full access to their private data. It further sees all messages in the system. But we do limit the adversary in two ways. First, we assume that it is computationally bounded so that we may use the PKI. Second, as proven in [1] for the synchronous model, achieving sub-quadratic complexity is impossible when the adversary can perform after-the-fact removal, meaning that it can delete messages that were sent by correct processes before corrupting these processes. Here, we adapt the no after-the-fact removal assumption to the asynchronous model, and define a *delayed-adaptive adversary* based on causality [26].

We formalize the concept of VRF-based committee sampling as used in Algorand [20, 15], and adapt it to the asynchronous model. In a nutshell, the idea is to use a VRF seeded with each process's private key in order to sample uniformly at random $O(\log n)$ processes for a *committee*, and to have different committees execute different parts of the BA protocol. Each committee is used for sending exactly one protocol message and messages are sent only by committee members, thus reducing the communication cost. Whereas in Algorand's synchronous model a process can be sure it receives messages from all correct committee members by a timeout, in the asynchronous model this is not the case. Rather, processes make progress by waiting for some threshold number of messages. Without committees, this threshold is normally $n-f$ (waiting for more than $n-f$ processes might violate termination). But since committees are randomly sampled, we do not know the committee's exact size or the number of Byzantine processes in it. Thus, adapting committees to this model is somewhat subtle and requires ensuring certain conditions regarding the intersection of subsets of committees. In this paper we identify sufficient conditions on sampling, which ensure safety and liveness with high probability.

Randomized BA algorithms can be seen as if processes toss a random coin at some point during the protocol. While some protocols toss a local coin [9, 12] and require exponential expected time to reach agreement, others use the abstraction of a *shared coin*, which involves communication among processes and results in the same coin toss with some well defined *success rate* [33, 14, 13, 20, 23]. In this work we present an asynchronous shared coin algorithm that uses a VRF and provides a constant success rate with an equal probability for tossing 0 and 1. Unlike previous shared coin implementations, our solution does not require a priori knowledge of the set of participants, which makes it useful in committee-based constructions. We then adapt our coin to work with committees and use it to devise a sub-quadratic BA algorithm.

In summary, this paper presents the first formalization of randomly sampled committees using cryptography in asynchronous settings. Based on this technique, it presents the first sub-quadratic asynchronous shared coin and BA whp algorithms. Our algorithms have expected $\widetilde{O}(n)$ word complexity and $O(1)$ expected time.

Roadmap. The rest of this paper is organized as follows. Section 2 describes the model; Section 3 reviews related work. In Section 4, we present our shared coin algorithm and in Section 5, we formalize committee sampling. Then, in Section 6, we use the coin and the committee sampling to construct a BA whp algorithm. We end with some concluding remarks in Section 7.

2 Model and Preliminaries

We consider a distributed system consisting of a well-known static set Π of n processes and a *delayed-adaptive adversary* (see definition below). The adversary may adaptively corrupt up to $f = (\frac{1}{3} - \epsilon)n$ processes in the course of a run, where $\max\{\frac{3}{8\ln n}, 0.109\} + \frac{1}{8\ln n} < \epsilon < \frac{1}{3}$. A corrupted process is *Byzantine*; it may deviate arbitrarily from the protocol. In particular, it may crash, fail to send or receive messages, and send arbitrary messages. As long as a process is not corrupted by the adversary, it is *correct* and follows the protocol.

Delayed-adaptive adversary. In the synchronous model, one defines a *late* adversary [34, 24, 7, 4], which at the beginning of round r, can observe the state of the system at the beginning of round $r-1$. This assumption prevents "after-the-fact" removals of messages sent by processes before being taken over by the adversary [1, 20], as required for achieving a sub-quadratic communication cost. We adapt this assumption to the asynchronous model. Since in asynchronous models the natural order between messages is Lamport's happens-before relation [26], we use the notion of causality instead of 'rounds' to define what messages the adversary may observe when scheduling other messages. We denote by $m \rightarrow m'$ the fact that m causally precedes m'. The adversary is formally defined as follows:

▶ **Definition 1** (delayed-adaptive adversary). *The delayed-adaptive adversary may adaptively corrupt up to f processes over the course of a run and schedules all messages. The adversary has full access to corrupted processes' private information and can observe all communication, but it can use the contents of a message m sent by a correct process for scheduling a message m' only if $m \rightarrow m'$.*

In addition, we assume that once the adversary takes over a process, it cannot "front run" messages that that process had already sent when it was correct, causing the correct messages to be supplanted. Blum et al. [10] achieve this property by using a separate key to encrypt each message, and deleting the secret key immediately thereafter.

Cryptographic tools. We assume a trusted PKI, where private and public keys for the processes are generated before the protocol begins and processes cannot manipulate their public keys. In addition, we assume that the adversary is computationally bounded, meaning that it cannot obtain the private keys of processes unless it corrupts them. Furthermore, we assume that the PKI is in place from the outset. (Recall that we assume a permissioned setting, so the public keys of the n processes are well-known). These assumptions allow us to use verifiable random functions, as we now define.

A *verifiable random function (*VRF*)* is a pseudorandom function that provides a proof of its correct computation [29]. Given a secret key sk, one can evaluate the VRF on any input x and obtain a pseudorandom output y together with a proof π, i.e., $\langle y, \pi \rangle = \text{VRF}_{sk}(x)$. From π and the corresponding public key pk, one can verify that y is correctly computed from x and sk using the function $\text{VRF-Ver}_{pk}(x, \langle y, \pi \rangle)$. Additionally, a VRF needs to satisfy *uniqueness*. More formally, a VRF guarantees the following properties:

- Pseudorandomness: for any x, it is infeasible to distinguish $y = \text{VRF}_{sk}(x)$ from a uniformly random value without access to sk.
- Verifiability: $\text{VRF-Ver}_{pk}(x, \text{VRF}_{sk}(x)) = \mathit{true}$.
- Uniqueness: it is infeasible to find $x, y_1, y_2, \pi_1, \pi_2$ such that $y_1 \neq y_2$ but $\text{VRF-Ver}_{pk}(x, \langle y_1, \pi_1 \rangle) = \text{VRF-Ver}_{pk}(x, \langle y_2, \pi_2 \rangle) = \mathit{true}$.

Efficient constructions for VRFs have been described in the literature [16, 19].

Communication. We assume that every pair of processes is connected via a reliable link. Messages are authenticated in the sense that if a correct process p_i receives a message m indicating that m was sent by a correct process p_j, then m was indeed generated by p_j and sent to p_i. The network is asynchronous, i.e., there is no bound on message delays.

Complexity. We use the following standard complexity notions [3, 30]. While measuring complexity, we allow a *word* to contain a signature, a VRF output, or a value from a finite domain. We define the *duration* of an execution as the longest sequence of messages that are causally related in this execution until all correct processes decide. We measure the expected *word communication complexity* of our protocols as the maximum of the expected total number of words sent by correct processes and the expected *running time* of our protocol as the maximum of the expected duration. In both cases the maximum is computed over all inputs and applicable adversaries and expectation is taken over the random VRF outputs.

3 Related Work

Lower bounds. Our assumptions conform with a number of known bounds. Deterministic consensus is impossible in an asynchronous system if even one process may crash (by FLP [18]) and requires $\Omega(n^2)$ communication even in synchronous systems [17]. As for randomized Byzantine Agreement, Abraham et al. [1] state that disallowing after-the-fact removal is necessary even in synchronous settings for achieving sub-quadratic communication.

Asynchronous BA and shared coin algorithms. The algorithms we present in this paper belong to the family of asynchronous BA algorithms, which sacrifice determinism in order to circumvent FLP. We compare our solutions to existing ones in Table 1.

Ben-Or [9] suggested a protocol with resilience $n > 5f$. This protocol uses a local coin (namely, a local source of randomness) and its expected time complexity is exponential (or constant if $f = O(\sqrt{n})$). Bracha [11] improved the resilience to $n > 3f$ with the same complexity. The complexity can be greatly reduced by replacing the local coin with a shared one with a guaranteed success rate.

Later works presented the shared coin abstraction and used it to solve BA with $O(n^2)$ communication. Rabin [33] was the first to do so, suggesting a protocol with resilience $n > 10f$ and a constant expected number of rounds. Cachin et al. [13] were the first to use a shared coin to solve BA with $O(n^2)$ communication and optimal resilience. Mostefaoui et al. [30] then presented a signature-free BA algorithm with optimal resilience and $O(n^2)$ messages that uses a shared coin abstraction as a black box; the shared coin algorithm we provide in Section 4 can be used to instantiate this protocol. All of the aforementioned algorithms solve binary BA, where the processes' initial values are 0 and 1; a recent work solved multi-valued BA with the same $O(n^2)$ word complexity [3].

BA algorithms also differ in the cryptographic assumptions they make and the cryptographic tools they use. Rabin's coin [33] is based on cryptographic secret sharing [35]. Some later works followed suit, and used cryptographic abstractions such as threshold signatures [3, 13]. Other works forgo cryptography altogether and instead consider a full information model, where there are no restrictions on the adversary's computational power [14, 23]. In this model, the problem is harder, and existing works achieve very low resilience [23] ($n > 400f$) or high communication complexity [14]. In this paper we do use cryptographic primitives. We assume a computationally bounded adversary and rely on the abstraction of a VRF [29]. VRFs were previously used in blockchain protocols [20, 21, 5] and were also used by Micali [28] to construct a shared coin in the synchronous model.

Several works [2, 8, 25, 32, 36] solve BA with subquadratic complexity in the so-called optimistic case (or "happy path"), when communication is timely and a correct process is chosen as a "leader". In contrast, we focus on the worst-case asynchronous case.

Committees. We use committees in order to reduce the word complexity and allow each step of the protocol to be executed by only a fraction of the processes. King and Saia used a similar concept and presented the first sub-quadratic BA protocol in the synchronous model [22]. Algorand proposed a synchronous algorithm [20] (and later extended it to eventual synchrony [15]) where committees are sampled randomly using a VRF. Each process executes a local computation to sample itself to a committee, and hence the selection of processes does not require interaction among them. We follow this approach in this paper

Table 1 Asynchronous Byzantine Agreement algorithms.

Protocol	n >	Adversary	Word complexity	Termination	Safety
Ben-Or [9]	$5f$	adaptive	$O(2^n)$	w.p. 1	✓
Rabin [33]	$10f$	adaptive	$O(n^2)$	w.p. 1	✓
Bracha [11]	$3f$	adaptive	$O(2^n)$	w.p. 1	✓
Cachin et al. [13]	$3f$	adaptive	$O(n^2)$	w.p. 1	✓
King-Saia [23]	$400f$	adaptive	polynomial	whp	✓
MMR [30]	$3f$	adaptive	$O(n^2)$	w.p. 1	✓
Our protocol	$\approx 4.5f$	delayed-adaptive	$\tilde{O}(n)$	whp	whp

and adapt the technique to the asynchronous model.

Following initial publication of our work, Blum et al. [10] have also achieved subquadratic BA WHP under an adaptive adversary. Their assumptions are incomparable to ours – while they strengthen the adversary to remove the delayed adaptivity requirement, they also strengthen the trusted setup. Specifically, they use a trusted dealer to a priori determine the committee members, flip the shared coin, and share it among the committee members. In contrast, we use a peer-to-peer protocol to generate randomness, and require delayed adaptivity in order to prevent the adversary from tampering with this randomness. As in our protocol, setup has to occur once and may be used for any number of BA instances.

4 Shared Coin

We describe here an asynchronous protocol for a shared coin with a constant success rate against the delayed-adaptive adversary. We assume that for every $r \in \mathbb{N}$, shared_coin(r) is invoked by all correct processes and that the invocation of shared_coin(r) by some process p is causally independent of its progress at other processes. The definition of a shared coin is given below.

▶ **Definition 2** (Shared Coin). *A shared coin with success rate ρ is a shared object that generates an infinite sequence of binary outputs. For each execution of the procedure shared_coin(r) with $r \in \mathbb{N}$, all correct processes output b with probability at least ρ, for any value $b \in \{0, 1\}$.*

The pseudo-code for our shared coin is presented in Algorithm 1. Our protocol is composed of two phases of messages passing. Each process first samples the VRF with its private key and the protocol's argument in order to generate a random initial value. For brevity, we denote by VRF_i the VRF with p_i's private key. Using a VRF to generate a random initial value effectively weakens the adversary as Byzantine processes can neither choose their initial values nor equivocate. If a Byzantine process would try to act maliciously, the VRF proof would easily expose it and its message would be ignored.

In each phase of the protocol, each process sends one value to every other process. The receiver validates the received values using the VRF proofs, which are sent along with the values. We omit the proof validation from the code for clarity. After two phases of communication, each process chooses the minimum value it received in the second phase and outputs its least significant bit. We follow the concept of a common core, as presented by Attiya and Welch for the crash failure model [6], and argue that if a core of $f + 1$ correct processes hold the global minimum value at the end of phase 1, then by the end of the following phase all processes receive this value. We exploit the ϵ parameter in our resilience definition to bound the number of values held by $f+1$ correct processes. We show that this number is linear in n and hence with a constant positive probability, by the end of the second phase, all correct processes receive the global minimum among the VRF outputs and therefore produce the same output.

Algorithm 1 Protocol shared_coin(r): code for process p_i

```
 1: Initially $\mathit{first\text{-}set},\ \mathit{second\text{-}set} = \emptyset$
 2: $v_i \leftarrow VRF_i(r)$
 3: send $\langle \text{FIRST}, v_i \rangle$ to all processes

 4: upon receiving $\langle \text{FIRST}, v_j \rangle$ with valid $v_j$ from $p_j$ do
 5:     if $v_j < v_i$ then $v_i \leftarrow v_j$
 6:     $\mathit{first\text{-}set} \leftarrow \mathit{first\text{-}set} \cup \{j\}$
 7:     when $|\mathit{first\text{-}set}| = n - f$ for the first time
 8:         send $\langle \text{SECOND}, v_i \rangle$ to all processes

 9: upon receiving $\langle \text{SECOND}, v_j \rangle$ with valid $v_j$ from $p_j$ do
10:     if $v_j < v_i$ then $v_i \leftarrow v_j$
11:     $\mathit{second\text{-}set} \leftarrow \mathit{second\text{-}set} \cup \{j\}$
12:     when $|\mathit{second\text{-}set}| = n - f$ for the first time
13:         return $LSB(v_i)$
```

We now prove that the shared coin has a constant success rate. We say that a value v is *common* if at least $f+1$ correct processes receive v by the end of phase 1. Denote by c be the number of different common values. The next two lemmas give a lower bound on c and on the probability that the global minimum among the VRF outputs is common.

▶ **Lemma 3.** *In Algorithm 1, $c \geq \frac{9\epsilon}{1+6\epsilon}n$.*

Proof. In a given run of the algorithm, define a table T with n rows and n columns, where for each correct process p_i and each $0 \leq j \leq n-1$, $T[i,j] = 1$ iff p_i receives $\langle \text{FIRST}, v \rangle$ from p_j before sending the second message in line 8. Each row of a correct process contains exactly $n-f$ ones since it waits for $n-f$ $\langle \text{FIRST}, v \rangle$ messages (line 7). Each row of a faulty process is arbitrarily filled with $n-f$ ones and f zeros. Thus, the total number of ones in the table is $n(n-f)$ and the total number of zeros is nf. Let k be the number of columns with at least $2f+1$ ones. Because each column represents a value and out of the $2f+1$ ones at least $f+1$ represent correct processes that receive this value, $c \geq k$. Denote by x the number of ones in the remaining columns. Because each column has at most n ones we get:

$$x \geq n(n-f) - kn. \tag{1}$$

And because the remaining columns have at most $2f$ ones:

$$x \leq 2f(n-k). \tag{2}$$

Combining (1), (2) we get:

$$2f(n-k) \geq n(n-f) - kn$$

$$2fn - 2fk \geq n^2 - fn - kn$$

$$(n-2f)k \geq n^2 - 3fn$$

$$k \geq \frac{n(n-3f)}{n-2f}.$$

Because $f = (\frac{1}{3} - \epsilon)n$ we get:

$$c \geq k \geq \frac{n(n-3(\frac{1}{3}-\epsilon)n)}{n-2(\frac{1}{3}-\epsilon)n} = \frac{n(1-1+3\epsilon)}{1-\frac{2}{3}+2\epsilon} = \frac{9\epsilon}{1+6\epsilon}n, \text{as required.}$$

◀

Let $v_{min} \triangleq \min_{p_i \in \Pi}\{VRF_i(r)\}$. We prove that with a constant probability, it is common.

▶ **Lemma 4.** *$Prob[v_{min}$ is common$] \geq \frac{c}{n} - \frac{1}{3} + \epsilon$.*

Proof. Notice that we assume that the invocation of shared_coin(r) by each process is causally independent of its progress at other processes. Hence, for any two processes p_i, p_j, the messages $\langle \text{FIRST}, v_i\rangle$, $\langle \text{FIRST}, v_j\rangle$ are causally concurrent. Thus, due to our *delayed-adaptive adversary* definition, these messages are scheduled by the adversary regardless of their content, namely their VRF random values. Notice that the adversary can corrupt processes before they initially send their VRF values. Since the adversary cannot predict the VRF outputs of the processes, the probability that the process holding v_{min} is corrupted before sending its FIRST messages is at most $\frac{f}{n}$. The adversary is oblivious to the correct processes' VRF values when it schedules their first phase messages. Therefore, each of them has the same probability to become common. Since at most f common values originate at Byzantine processes, this probability is at least $\frac{c-f}{n-f}$. We conclude that v_{min} is common with probability at least $(1-\frac{f}{n})\frac{c-f}{n-f} = (1-\frac{(\frac{1}{3}-\epsilon)n}{n})\frac{c-(\frac{1}{3}-\epsilon)n}{n-(\frac{1}{3}-\epsilon)n} = (\frac{2}{3}+\epsilon)\frac{c-(\frac{1}{3}-\epsilon)n}{(\frac{2}{3}+\epsilon)n} = \frac{c-(\frac{1}{3}-\epsilon)n}{n} = \frac{c}{n} - \frac{1}{3} + \epsilon$. ◀

We next observe that if v_{min} is common, then it is shared by all processes.

▶ **Lemma 5.** *If v_{min} is common then each correct process holds v_{min} at the end of phase 2.*

Proof. Since v_{min} is common, at least $f+1$ correct processes receive it by the end of phase 1 and update their local values to v_{min}. During the second phase, each correct process hears from $n-f$ processes. This means that it hears from at least one correct process that has updated its value to v_{min} and sent it. ◀

▶ **Lemma 6.** *The coin's success rate is at least $\frac{18\epsilon^2+24\epsilon-1}{6(1+6\epsilon)}$.*

Proof. We bound the probability that all correct processes output $b \in \{0,1\}$ as follows:

$Prob$[all correct processes output b] $\geq Prob$[all correct processes have the same v_i at the end of phase 2 and its LSB is b] $\geq Prob$[all correct processes have $v_i = v_{min}$ at the end of phase 2 and its LSB is b] $= \frac{1}{2} \cdot Prob$[all correct processes have $v_i = v_{min}$ at the end of phase 2] $\overset{\text{Lemma 5}}{\geq} \frac{1}{2} \cdot Prob[v_{min}$ is common$] \overset{\text{Lemma 4}}{\geq} \frac{1}{2}(\frac{c}{n} - \frac{1}{3} + \epsilon) \overset{\text{Lemma 3}}{\geq} \frac{18\epsilon^2+24\epsilon-1}{6(1+6\epsilon)}$.

◀

▶ Remark 7. Notice that for $\epsilon = \frac{1}{3}$ (i.e., $f = 0$) it holds that the coin's success rate is $\frac{1}{2}$ and we get a perfect fair coin.

We have shown a bound on the coin's success rate in terms of ϵ. Since $\epsilon > 0.109$, the coin's success rate is a positive constant. We next prove that the coin ensures liveness.

▶ **Lemma 8.** *If all correct processes invoke Algorithm 1 then all correct processes return.*

Proof. All correct processes send their messages in the first phase. As up to f processes may be faulty, each correct process eventually receives $n-f$ $\langle \textsc{first}, x\rangle$ messages and sends a message in the second phase. As $n-f$ correct processes send their messages, each correct process eventually receives $n-f$ $\langle \textsc{second}, x\rangle$ messages and returns. ◀

From Lemma 6 and Lemma 8 we conclude:

▶ **Theorem 9.** *Algorithm 1 implements a shared coin with success rate at least* $\frac{18\epsilon^2+24\epsilon-1}{6(1+6\epsilon)}$.

Complexity. In each shared coin instance all correct processes send two messages to all other processes. Each of these messages contains one VRF output (including a value and a proof), in addition to a constant number of bits that identify the message's type. Therefore, each message's size is a constant number of words and the total word complexity of a shared coin instance is $O(n^2)$.

We have presented a new shared coin in the asynchronous model that uses a VRF. This coin can be incorporated into the Byzantine Agreement algorithm of Mostefaoui et al. [30], to yield an asynchronous binary Byzantine Agreement with resilience $f = (\frac{1}{3}-\epsilon)n$, a word complexity of $O(n^2)$, and $O(1)$ expected time.

5 Committees

5.1 Validated committee sampling

With the aim of reducing the number of messages and achieving sub-quadratic word complexity, it is common to avoid all-to-all communication phases [20, 22]. Instead, a subset of processes is sampled to a committee and only processes elected to the committee send messages. As committees are randomly sampled, preventing the adversary from corrupting their members, each committee member cannot predict the next committee sample and send its message to all other processes. Potentially, if the committee is sufficiently small, this technique allow committee-based protocols to result in sub-quadratic word complexity.

Using VRFs, it is possible to implement *validated committee sampling*, which is a primitive that allows processes to elect committees without communication and later prove their election. It provides every process p_i with a private function $\mathit{sample}_i(s,\lambda)$, which gets a string s and a threshold $1 \le \lambda \le n$ and returns a tuple $\langle v_i, \sigma_i\rangle$, where $v_i \in \{\mathit{true}, \mathit{false}\}$ and σ_i is a proof that $v_i = \mathit{sample}_i(s,\lambda)$. If $v_i = \mathit{true}$ we say that p_i is *sampled* to the committee for s and λ. The primitive ensures that p_i is sampled with probability $\frac{\lambda}{n}$. In addition, there is a public (known to all) function, $\mathit{committee\text{-}val}(s,\lambda,i,\sigma_i)$, which gets a string s, a threshold λ, a process identification i and a proof σ_i, and returns *true* or *false*.

Consider a string s. For every i, $1 \le i \le n$, let $\langle v_i, \sigma_i\rangle$ be the return value of $\mathit{sample}_i(s,\lambda)$. The following is satisfied for every p_i:

- $\mathit{committee\text{-}val}(s,\lambda,i,\sigma_i) = v_i$.
- If p_i is correct, then it is infeasible for the adversary to compute $\mathit{sample}_i(s,\lambda)$.
- It is infeasible for the adversary to find $\langle v, \sigma\rangle$ s.t. $v \ne v_i$ and $\mathit{committee\text{-}val}(s,\lambda,i,\sigma) = \mathit{true}$.

We refer to the set of processes sampled to the committee for s and λ as $C(s,\lambda)$. In this paper we set λ to $8\ln n$. Let d be a parameter of the system such that $\max\{\frac{1}{\lambda}, 0.0362\} < d < \frac{\epsilon}{3} - \frac{1}{3\lambda}$. Our committee-based protocols can no longer wait for $n-f$ processes. Instead, they wait for $W \triangleq \lceil (\frac{2}{3} + 3d)\lambda \rceil$ processes. We show that whp at least W processes will be correct in each committee sample and hence waiting for this number does not compromise liveness. In addition, instead of assuming f Byzantine processes, our committee-based protocols assume that whp the number of Byzantine processes in each committee is at most $B \triangleq \lfloor (\frac{1}{3} - d)\lambda \rfloor$. The following claim is proven in Appendix A using Chernoff bounds.

▷ Claim 10. For a string s and $\lambda = const \cdot \ln n$ the following hold with high probability:

(S1) $|C(s,\lambda)| \leq (1+d)\lambda$.
(S2) $|C(s,\lambda)| \geq (1-d)\lambda$.
(S3) At least W processes in $C(s,\lambda)$ are correct.
(S4) At most B processes in $C(s,\lambda)$ are Byzantine.

If a protocol uses a constant number of committees, then with high probability, Claim 10 holds for all of them. If, however, a protocol uses a polynomial number of committees then it does not guarantee the properties of this claim. The following corollaries are derived from Claim 10 and are used to ensure the safety and liveness properties of our protocols that use committees (a full proof is in Appendix A). Intuitively, S3 allows the protocol to wait for W messages without forgoing liveness. Property S5 below shows that if two processes wait for sets P_1 and P_2 of this size, then they hear from at least $B+1$ common processes of which, by S4, at least one is correct.

▶ **Corollary 11** (S5). *Consider $C(s,\lambda)$ for some string s and some $\lambda = const \cdot \ln n$ and two sets $P_1, P_2 \subset C(s,\lambda)$ s.t $|P_1| = |P_2| = W$. Then, $|P_1 \cap P_2| \geq B+1$.*

The following property is used to show that if $B+1$ correct processes hold some value, and some correct process waits for messages from W processes, then it hears from at least one correct process that holds this value.

▶ **Corollary 12** (S6). *Consider $C(s,\lambda)$ for some string s and some $\lambda = const \cdot \ln n$ and two sets $P_1, P_2 \subset C(s,\lambda)$ s.t $|P_1| = B+1$ and $|P_2| = W$. Then, $|P_1 \cap P_2| \geq 1$.*

5.2 WHP Coin

We now employ committee sampling to reduce the word complexity of our shared coin. Our new protocol is called whp_coin. As before, we assume that for every $r \in \mathbb{N}$, the invocation of whp_coin(r) by some process p is causally independent of its progress at other processes. We now define the *WHP coin* abstraction:

▶ **Definition 13** (WHP Coin). *A WHP coin with success rate ρ is a shared object exposing whp_coin(r), $r \in \mathbb{N}$ at each process. If all correct processes invoke whp_coin(r) then, whp (1) all correct processes return, and (2) all of them output the same value b with probability at least ρ, for any value $b \in \{0,1\}$.*

The whp_coin protocol is presented in Algorithm 2. It samples two committees, one for each communication step. In each step, only the processes that are sampled to the committee send messages. However, since the committee samples are unpredictable, messages are sent to all processes. With committees, processes can no longer wait for $n-f$ messages. Instead they wait for W messages. Since a constant number of committees is sampled in the protocol, Claim 10 holds for all of them and by S3, all processes receive W messages, ensuring liveness.

In Appendix B we adapt the coin's correctness proof given in Section 4 to the committee-based protocol, proving the following theorem:

Algorithm 2 Protocol whp_coin(r): code for process p_i

```
Initially $first\text{-}set, second\text{-}set = \emptyset$, $v_i = \infty$
if $sample_i(\text{FIRST}, \lambda) = true$ then
    $v_i \leftarrow VRF_i(r)$
    send $\langle \text{FIRST}, v_i \rangle$ to all processes

upon receiving $\langle \text{FIRST}, v_j \rangle$ with valid $v_j$
      from validly sampled $p_j$ do
    if $sample_i(\text{SECOND}, \lambda)$ then
        if $v_j < v_i$ then $v_i \leftarrow v_j$
        $first\text{-}set \leftarrow first\text{-}set \cup \{j\}$
        when $|first\text{-}set| = W$ for the first time
            send $\langle \text{SECOND}, v_i \rangle \rangle$ to all processes

upon receiving $\langle \text{SECOND}, v_j \rangle$ with valid $v_j$
      from validly sampled $p_j$ do
    if $v_j < v_i$ then $v_i \leftarrow v_j$
    $second\text{-}set \leftarrow second\text{-}set \cup \{j\}$
    when $|second\text{-}set| = W$ for the first time
        return $LSB(v_i)$
```

▶ **Theorem 14.** *Algorithm 2 implements a WHP coin with a constant success rate.*

Complexity. In each whp_coin instance using committees all correct processes that are sampled to the two committees (lines 2,6) send messages to all other processes. Each of these messages contains a VRF output (including a value and a proof), a VRF proof of the sender's election to the committe and a constant number of bits that identify the type of message that is sent. Therefore, each message's size is a constant number of words and the total word complexity of a WHP coin instance is $O(nC)$ where C is the number of processes that are sampled to the committees. Since each process is sampled to a committee with probability $\frac{1}{\lambda}$, we get a word complexity of $O(n\lambda) = O(n \log n) = \widetilde{O}(n)$ in expectation.

6 Asynchronous sub-quadratic Byzantine Agreement

We adapt the Byzantine Agreement algorithm of Mostefaoui et al. [30] to work with committees. Our protocol leverages an *approver* abstraction, which we implement in Section 6.1 and then integrate it into a Byzantine Agreement protocol in Section 6.2.

6.1 Approver abstraction

The *approver* abstraction is an adaptation of the Synchronized Binary-Value Broadcast (SBV-broadcast) primitive in [30]. It provides processes with the procedure *approve(v)*, which takes a value v as an input and returns a set of values.

▶ **Assumption 1.** *Correct processes invoke the approver with at most 2 different values.*

Under this assumption, an approver satisfies the following:

▶ **Definition 15** (Approver). *In an approver instance the following properties hold whp:*

- *Validity. If all correct processes invoke approve(v) then the only possible return value of correct processes is* $\{v\}$.
- *Graded Agreement. If a correct process* p_i *returns* $\{v\}$ *and another correct process* p_j *returns* $\{w\}$ *then* $v = w$.
- *Termination. If all correct processes invoke approve then approve returns with a non-empty set at all of them.*

Our approver uses different committees for different message types, as illustrated in Fig. 1. Importantly, the protocol satisfies the so-called *process replaceability* [20] property, whereby a correct process selected for a committee C broadcasts at most one message in its role as a member of C. Thus, our delayed-adaptive adversary can learn of a process's membership in a committee only after that process ceases to partake in the committee. This allows us to leverage the sampling analysis in the previous section. For clarity of the presentation, we discuss the algorithm here under the assumption that properties S1-S6 hold for all sampled committees. As shown above, these hold whp for each committee, and the algorithm employs a constant number of committees, so they hold for all of them whp.

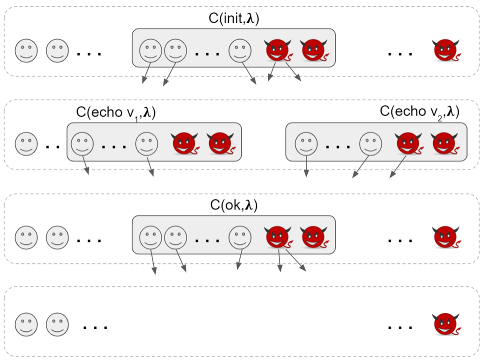

Figure 1 Committees sampled in Algorithm 3.

The approver's pseudo-code appears in Algorithm 3. It consists of three phases – init, echo, and ok. In each phase, committee members broadcast to all processes. Messages are validated to originate from legitimate committee members using the *committee-val* primitive; this validation is omitted from the pseudo-code for clarity. In the first phase, each init committee member broadcasts its input value to all processes.

The role of the echo phase is to "boost" values sent by sufficiently many processes in the init phase, and make sure that *all* correct processes receive them. "Sufficiently many" here means at least $B + 1$, which by S4 includes at least one correct process. Ensuring process replaceability in the echo phase is a bit tricky, since committee members must echo every value they receive from least $B + 1$ processes, and there might be two such values. (Recall that we assume that correct processes invoke the protocol with at most two different values, so there cannot be more than two values that exceed this threshold). To ensure that each committee member broadcasts at most once, we sample a different committee for each value. That is, the value v is part of the string passed to the sample function for this phase.

When a member of the ok committee receives $\langle \text{ECHO}, v\rangle$ messages from W different members of the same echo committee for the first time, it broadcasts an $\langle \text{OK}, v\rangle$ message. Note that the process sends an ok message only for the first value that exceeds this threshold. An $\langle \text{OK}, v\rangle$ message includes, as proof of its validity, W signed $\langle \text{ECHO}, v\rangle$ messages. Again, the proof and its validation are omitted from the pseudo-code for clarity. Once a correct process receives W valid OK messages, it returns the set of values in these messages.

Algorithm 3 Protocol approve(v_i): code for process p_i

```
1: if $sample_i(\text{INIT}, \lambda) = true$ then broadcast $\langle \text{INIT}, v_i \rangle$

2: upon receiving $\langle \text{INIT}, v \rangle$ from $B + 1$ different processes do
3:     if $sample_i(\langle \text{ECHO}, v \rangle, \lambda) = true$ then broadcast $\langle \text{ECHO}, v \rangle$

4: upon receiving $\langle \text{ECHO}, v \rangle$ from $W$ different processes do
5:     if $sample_i(\text{OK}, \lambda) = true \wedge$ haven't sent any $\langle \text{OK}, * \rangle$ message then
6:         broadcast $\langle \text{OK}, v \rangle$

7: upon receiving $\langle \text{OK}, * \rangle$ from $W$ different processes do
8:     return the set of values received in these messages
```

We next prove that Algorithm 3 implements an approver.

▶ **Lemma 16** (Validity). *If all correct processes invoke* approve(v) *then the only possible return value of correct processes is* $\{v\}$ *whp.*

Proof. By Claim 10 S4 holds whp for the four sampled committees. It remains to show that S4 implies validity. Since by S4 the number of Byzantine processes sampled to the init committee in line 1 is at most B, no process receives $B + 1$ messages with a value $w \neq v$. Thus, no correct process echoes $\langle \text{ECHO}, w \rangle$ in line 3. Because the number of Byzantine processes in $C(\langle \text{ECHO}, w \rangle, \lambda)$ in line 3 is also at most B, no correct process receives more than B $\langle \text{ECHO}, w \rangle$ messages. As a result, since $B < W$, no $\langle \text{OK}, w \rangle$ message is sent by any correct process. Since ok messages carry proofs, no Byzantine process can send a valid $\langle \text{OK}, w \rangle$ either. Therefore, the only possible value in the ok messages is v, and no other value is returned. ◀

▶ **Lemma 17** (Graded Agreement). *If a correct process p_i returns $\{v\}$ and another correct process p_j returns $\{w\}$ then $v = w$ whp.*

Proof. By Claim 10 and Corollary 11, S4 and S5 hold whp for the four sampled committees. We show that S4 and S5 imply graded agreement. Assume p_i returns $\{v\}$ and p_j returns $\{w\}$. Then p_i receives W $\langle \text{OK}, v \rangle$ messages and p_j receives W $\langle \text{OK}, w \rangle$ messages. By S5, two sets of size W intersect by at least $(\frac{1}{3} - d)\lambda + 1$ processes. Hence, since by S4 there are at most B Byzantine processes in the ok committee, there is at least one correct process p_k whose ok message is received by both p_i and p_j whp. It follows that p_k sends $\langle \text{OK}, v \rangle$ and $\langle \text{OK}, w \rangle$. Since every correct process sends at most one ok message (line 5), $v = w$. ◀

▶ **Lemma 18** (Termination). *If all correct processes invoke approve then at every correct process approve returns with a non-empty set whp.*

Proof. By Claim 10 S3 holds whp. We show that S3 implies termination. Because all correct processes invoke approve, every correct init committee member in line 1 sends $\langle \text{INIT}, v_i \rangle$. Notice that $\frac{1}{2}W > (\frac{1}{3} - d)\lambda \geq B$. Hence, since the number of correct processes in the init committee is at least W (S3) and correct processes may send at most two different initial values (Assumption 1), one of them is sent by at least $B + 1$ correct processes. Denote this value by v. Every correct process receives this value from $B+1$ processes, and if it is sampled to $C(\langle \text{ECHO}, v\rangle, \lambda)$ in line 3 then it sends it to all other processes. Since $C(\langle \text{ECHO}, v\rangle, \lambda)$ also has at least W correct processes (S3), every correct process p receives W $\langle \text{ECHO}, v\rangle$ messages. If p is sampled to the ok committee in line 5 and at this point p has not yet sent an $\langle \text{OK}, *\rangle$ message, it sends one. Since there are at least W correct processes that are sampled to the ok committee (S3) and they all send OK messages (possibly for different values), every correct process receives W OK messages and returns the non-empty set of approved values. ◀

From Lemmas 16,17,18, we conclude the following theorem:

▶ **Theorem 19.** *Algorithm 3 implements an approver.*

Complexity. In each approver instance correct processes that are sampled to the four committees (lines 1,3,5) send messages to all other processes. The committee size is $O(\lambda) = O(\log n)$ whp. Messages contain values, VRF proofs of the sender's election to the committee, signatures of $O(\lambda)$ committee members, and a constant number of bits that identify the type of message that is sent. Therefore, each message's size is at most $O(\lambda)$ words and the total word complexity of a shared coin instance is $O(n\lambda^2) = O(n \log^2 n) = \widetilde{O}(n)$ in expectation. The λ^2 appears in the expression due to the signatures of $O(\lambda)$ processes sent along the ok messages.

6.2 Byzantine Agreement WHP

Our next step is solving Byzantine Agreement whp, formally defined as follows:

▶ **Definition 20** (Byzantine Agreement WHP). *In Byzantine Agreement WHP, each correct process $p_i \in \Pi$ proposes a binary input value v_i and decide on an output value $decision_i$ s.t. with high probability the following properties hold:*

- *Validity. If all correct processes propose the same value v, then any correct process that decides, decides v.*
- *Agreement. No two correct processes decide differently.*
- *Termination. Every correct process eventually decides.*

We present the pseudo-code for our algorithm in Algorithm 4. Our protocol executes in rounds. Each round consists of two approver invocations and one call to the WHP coin. Again, we discuss the algorithm assuming S1-S6 hold. We will argue that the algorithm decides in a constant number of rounds whp, and so these properties hold for all the committees it uses. The local variable est_i holds p_i's current estimate of the decision value. The variable $decision_i$ holds p_i's irrevocable decision. It is initialized to $\perp$ and set to a value in $\{0,1\}$ at most once. Every process p_i begins by setting est_i to hold its initial value. At the beginning of each round processes execute the approver with their est values. If they return a singleton $\{v\}$, they choose to invoke the next approver with v as their proposal and otherwise they invoke the next approver with $\perp$. By the approver's graded agreement property, different processes do not return different singletons. Thus, at most two different values ($\perp$ and one in $\{0,1\}$) are given as an input by correct processes to the next approver, satisfying Assumption 1.

At this point, after all correct processes have chosen their proposals, they all invoke the WHP coin in line 8 in order to select a fall-back value. Notice that executing the WHP coin protocol after proposals have been set prevents the adversary from biasing proposals based on the coin flip. Then, in in line 9, all processes invoke the approver with their proposals. If a process does not receive $\perp$ in its return set, it can safely decide the value it received. It does so by updating its *decision* variable in line 13. If it receives some value other than $\perp$ it adopts it to be its estimated value (line 18), whereas if it receives only $\perp$, it adopts the coin flip (line 16). If all processes receive $\perp$ in line 4 then the probability that they all adopt the same value is at least 2ρ, where ρ is the coin's success rate. If some processes receive v, then the probability that all the processes that adopt the coin flip adopt v is at least ρ. With high probability, after a constant number of rounds, all correct processes have the same estimated value. By validity of the approver, once they all have common estimate, they decide upon it within 1 round.

Algorithm 4 Protocol Byzantine Agreement(v_i): code for process p_i

```
1:  $est_i \leftarrow v_i$
2:  $decision_i \leftarrow \perp$

3:  for $r = 0, 1, \ldots$ do
4:      $vals \leftarrow$ approve($est_i$)
5:      if $vals = \{v\}$ for some $v$ then
6:          $propose_i \leftarrow v$
7:      otherwise $propose_i \leftarrow \perp$

8:      $c \leftarrow$ whp_coin($r$)
9:      $props \leftarrow$ approve($propose_i$)
10:     if $props = \{v\}$ for some $v \neq \perp$ then
11:         $est_i \leftarrow v$
12:         if $decision_i = \perp$ then
13:             $decision_i \leftarrow v$
14:     else
15:         if $props = \{\perp\}$ then
16:             $est_i \leftarrow c$
17:         else                                  ▷ $props = \{v, \perp\}$
18:             $est_i \leftarrow v$
```

We now prove our main theorem:

▶ **Theorem 21.** *Algorithm 4 when using an approver (Definition 15) and a WHP coin (Definition 13) solves Byzantine Agreement whp (Definition 20).*

We first show that Algorithm 4 satisfies the approver and WHP coin primitives' assumptions whp. Proving this allows us to use their properties while proving Theorem 21.

▶ **Lemma 22.** *For every round r of Algorithm 4 the following hold:*
1. *All correct processes invoke approve with at most 2 different values.*
2. *The invocation of whp_coin(r) by a correct process p is causally independent of its progress at other processes.*

Proof.
1. It is easy to see, by induction on the number of rounds, that since the processes' inputs are binary and we use a binary coin, the *est* of all processes is in $\{0, 1\}$ at the beginning of each round. Hence, the approver in line 4 is invoked with at most two different values. Due to its graded agreement property, all processes that update their propose to $v \neq \perp$ in line 6 update it to the same value whp. Thus, whp, in line 9 approver is invoked with either v or $\perp$.
2. Correct processes call *whp_coin*(r) without waiting for indication that other processes have done so.

◀

Next, we show that for any given round of the algorithm, (1) whp all processes complete this round, and (2) with a constant probability, they all have the same estimate value by its end.

▶ **Lemma 23.** *If all correct processes begin round r of Algorithm 4 then whp:*

1. *All correct processes complete round r, i.e. they're not blocked during round r.*
2. *With probability greater than ρ, where ρ is the success rate of the WHP coin, all correct processes have the same est value at the end of round r.*

Proof. First, if all correct processes begin round r then they all invoke the approver in line 4. Their invocation returns whp so they all invoke the coin in line 8, and so it returns and all invoke approve in line 9, and so it also returns, proving (1). To show (2), consider the possible scenarios with respect to the approver's return value:

- All correct processes return singletons in line 4:
  By the approver's graded agreement, whp they return $\{v\}$ with the same value v. Hence, all correct processes update their *propose* to v. Then, they all execute approve(v) in line 9, and by validity, they all return $\{v\}$ whp. In this case they all update $est \leftarrow v$.
- All correct processes return $\{0, 1\}$ in line 4:
  All correct processes update their *propose* value to $\perp$. Then, they all execute approve($\perp$) in line 9, and by validity, they return $\{\perp\}$ whp. In this case, all correct processes then update their estimate value to the coin flip (line 16). With probability at least 2ρ all correct processes toss the same $v \in \{0, 1\}$.
- Some, but not all correct processes return singletons in line 4:
  By graded agreement, all singletons hold the same value v. Thus, all correct processes propose v or $\perp$ and by validity return $\{v\}, \{v, \perp\}$, or $\{\perp\}$ in line 9. We examine two possible complementary sub-cases:

  - If some correct process returns $\{v\}$ in line 9: By approver's graded agreement, no correct process returns $\{\perp\}$ in line 9, whp. Thus, whp, all correct processes update their estimate value to v (in line 11 or 18).
  - If no correct process return $\{v\}$ in line 9: All correct processes returns $\{v, \perp\}$ or $\{\perp\}$ in line 9. All correct processes either update their estimate value to the coin flip of the WHP coin (line 16) or to v (line 18). Since the value v is determined before tossing the coin, the adversary cannot bias v after viewing the coin flip and with probability at least ρ all correct processes that adopt the coin's value toss v.

In all cases, with probability greater than ρ all correct processes have the same *est* value at the end of r, whp. ◀

The following lemmas indicate that the Byzantine Agreement whp properties are satisfied, which completes the proof of Theorem 21.

▶ **Lemma 24.** *(Validity) If at the beginning of round r of Algorithm 4 all correct processes have the same estimate value v, then whp any correct process that has not decided before decides v in round r.*

Proof. If all correct processes start round r then by Lemma 23 they all complete round r. Since they all being with the same estimate value v, they all execute approve(v) in line 4. Hence, by approver's validity and termination, whp they all return the non-empty set $\{v\}$ and update their *propose* values to v. Then, they all execute approve(v) for the second time in line 9, and due to the same reason, they all return $\{v\}$ whp. Any correct process that has not decided before decides v in line 13. ◀

▶ **Lemma 25.** *(Termination) Every correct process decides whp.*

Proof. By Lemma 23, for every round r of Algorithm 4, with probability greater than ρ, where ρ is the success rate of the WHP coin, all correct processes have the same *est* value at the end of r whp. Hence, by Lemma 24, with probability greater than ρ, all correct processes decide by round $r+1$ whp. It follows that the expected number of rounds until all processes decide is bounded by $\frac{1}{\rho}$, which is constant. Thus, by Chebyshev's inequality, whp all correct processes decide within a constant number of rounds. ◀

▶ **Lemma 26.** *(Agreement) No two correct processes decide different values whp.*

Proof. Let r be the first round in which some process p_i decides on some value $v \in \{0,1\}$. Thus, p_i's invocation to approver in line 9 of round r returns $\{v\}$. If another correct process p_j decides w in round r then its approver call in line 9 of round r returns $\{w\}$. By approver's graded agreement, $v = w$ whp. Consider a correct process p_k that does not decide in round r. By the definition of r, p_k hasn't decided in any round $r' < r$. By approver's graded agreement, whp, p_k returns $\{v, \perp\}$ in line 9 of round r, and p_k updates its est_k value to v in line 18. It follows that whp all correct processes have v as their estimate value at the beginning of round $r+1$. By Lemma 24, every correct process that has not decided in round r decides v in round $r+1$ whp. ◀

Complexity. In each round of the protocol, all correct processes invoke two approver calls and one WHP coin instance. Due to the constant success rate of the WHP coin, the expected number of rounds before all correct processes decide is constant. Thus, due to the word complexity of the WHP coin and approver, the expected word complexity is $O(n \log^2 n) = \widetilde{O}(n)$ and the time complexity is $O(1)$ in expectation.

7 Conclusions and Future Directions

We have presented the first sub-quadratic asynchronous Byzantine Agreement algorithm. To construct the algorithm, we introduced two techniques. First, we presented a shared coin algorithm that requires a trusted PKI and uses VRFs. Second, we formalized VRF-based committee sampling in the asynchronous model for the first time.

Our algorithm solves Byzantine Agreement with high probability. It would be interesting to understand whether some of the problem's properties can be satisfied with probability 1, while keeping the sub-quadratic communication cost. In addition, in order to achieve the constant success rate of the coin and guarantee the committees' properties, we bounded ϵ from below by a constant. This bound prevented us from achieving optimal resilience. The question whether it is possible to relax this bound to allow better resilience remains open.

Acknowledgements

We thank Ittai Abraham, Dahlia Malkhi, Kartik Nayak and Ling Ren for insightful initial discussions.

References

1 Ittai Abraham, TH Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication complexity of byzantine agreement, revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 317–326, 2019.

2 Ittai Abraham, Guy Golan-Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message bft devil. *CoRR*, abs/1803.05069, 2018.

3 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*, pages 337–346, 2019.

4 Mohamad Ahmadi, Abdolhamid Ghodselahi, Fabian Kuhn, and Anisur Rahaman Molla. The cost of global broadcast in dynamic radio networks. *Theoretical Computer Science*, 806:363–387, 2020.

5 Avi Asayag, Gad Cohen, Ido Grayevsky, Maya Leshkowitz, Ori Rottenstreich, Ronen Tamari, and David Yakira. Helix: A scalable and fair consensus algorithm resistant to ordering manipulation. *IACR Cryptology ePrint Archive*, 2018:863, 2018.

6 Hagit Attiya and Jennifer Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. John Wiley & Sons, 2004.

7 Baruch Awerbuch and Christian Scheideler. A denial-of-service resistant dht. In *International Symposium on Distributed Computing*, pages 33–47. Springer, 2007.

8 Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

9 Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

10 Erica Blum, Jonathan Katz, Chen-Da Liu-Zhang, and Julian Loss. Asynchronous byzantine agreement with subquadratic communication. Cryptology ePrint Archive, Report 2020/851, 2020.

11 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

12 Gabriel Bracha and Sam Toueg. Resilient consensus protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 12–26. ACM, 1983.

13 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

14 Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *STOC*, volume 93, pages 42–51. Citeseer, 1993.

15 Jing Chen, Sergey Gorbunov, Silvio Micali, and Georgios Vlachos. Algorand agreement: Super fast and partition resilient byzantine agreement. *IACR Cryptology ePrint Archive*, 2018:377, 2018.

16 Yevgeniy Dodis and Aleksandr Yampolskiy. A verifiable random function with short proofs and keys. In *International Workshop on Public Key Cryptography*, pages 416–431. Springer, 2005.

17 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, January 1985.

18 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

19 Matthew Franklin and Haibin Zhang. Unique ring signatures: A practical construction. In *International Conference on Financial Cryptography and Data Security*, pages 162–170. Springer, 2013.

20 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

21 Timo Hanke, Mahnush Movahedi, and Dominic Williams. Dfinity technology overview series, consensus system. *arXiv preprint arXiv:1805.04548*, 2018.

22 Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: scalable byzantine agreement with an adaptive adversary. *Journal of the ACM (JACM)*, 58(4):1–24, 2011.

23 Valerie King and Jared Saia. Byzantine agreement in polynomial expected time. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 401–410. ACM, 2013.

24 Marek Klonowski, Dariusz R Kowalski, and Jarosław Mirek. Ordered and delayed adversaries and how to work against them on a shared channel. *Distributed Computing*, 32(5):379–403, 2019.

25 Jae Kwon. Tendermint: Consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.

26 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

27 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

28 Silvio Micali. Very simple and efficient byzantine agreement. In Christos H. Papadimitriou, editor, *8th Innovations in Theoretical Computer Science Conference, ITCS 2017, January 9-11, 2017, Berkeley, CA, USA*, volume 67 of *LIPIcs*, pages 6:1–6:1. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ITCS.2017.6.

29 Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999.

30 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *Journal of the ACM (JACM)*, 62(4):31, 2015.

31 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

32 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *arXiv preprint arXiv:1909.05204*, 2019.

33 Michael O Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409. IEEE, 1983.

34 Peter Robinson, Christian Scheideler, and Alexander Setzer. Breaking the $\Omega(\sqrt{n})$ barrier: Fast consensus under a late adversary. In *30th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2018*, pages 173–182. ACM New York, 2018.

35 Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

36 Alexander Spiegelman. In search for an optimal authenticated byzantine agreement. In *35th International Symposium on Distributed Computing*, 2021.

Appendix A Sampling proofs

▷ Claim 27. For a string s and $\lambda = const \cdot \ln n$ the following hold with high probability:

(S1) $|C(s, \lambda)| \leq (1 + d)\lambda$.
(S2) $|C(s, \lambda)| \geq (1 - d)\lambda$.
(S3) At least W processes in $C(s, \lambda)$ are correct.
(S4) At most B processes in $C(s, \lambda)$ are Byzantine.

Proof. Recall that d is a parameter of the system such that $\max\{\frac{1}{\lambda}, 0.0362\} < d < \frac{\epsilon}{3} - \frac{1}{3\lambda}$.

In order to prove these properties we use two Chernoff bounds:

Suppose $X_1, ..., X_n$ are independent random variables taking values in $\{0, 1\}$. Let X denote their sum and let $E[X]$ denote the sum's expected value.

$$\forall 0 \leq \delta : \; Pr[X \geq (1 + \delta)E[X]] \leq e^{-\frac{\delta^2 E[X]}{2+\delta}} \tag{3}$$

$$\forall 0 \leq \delta \leq 1 : \; Pr[X \leq (1 - \delta)E[X]] \leq e^{-\frac{\delta^2 E[X]}{2}} \tag{4}$$

▶ **Lemma 28** (S1)**.** $|C(s, \lambda)| \leq (1 + d)\lambda$ *whp.*

Proof. Let X be a random variable that represents the number of processes that are sampled to $C(s, \lambda)$. $X \sim Bin(n, \frac{const \cdot \ln n)}{n})$, thus $E[X] = const \cdot \ln n$.

Placing $\delta = d \geq 0$ in 3 we get:

$$Pr[X \geq (1 + d)const \cdot \ln n] \leq e^{-\frac{d^2 const \cdot \ln n}{2+d}}.$$

Denote by c_1 the constant $\frac{const \cdot d^2}{2+d}$. We get:

$$Pr[X \geq (1 + d)const \cdot \ln n] \leq e^{-c_1 \ln n}.$$

Thus,

$$Pr[X < (1 + d)const \cdot \ln n] = Pr[X < (1 + d)\lambda] > 1 - \frac{1}{e^{c_1 \ln n}} = 1 - \frac{1}{n^{c_1}}.$$

◀

▶ **Lemma 29** (S2)**.** $|C(s, \lambda)| \geq (1 - d)\lambda$ *whp.*

Proof. Let X be a random variable that represents the number of processes that are sampled to $C(s, \lambda)$. $X \sim Bin(n, \frac{const \cdot \ln n}{n})$, thus $E[X] = const \cdot \ln n$.

Placing $\delta = d$ it holds that $0 \leq \delta \leq 1$ in 4 and we get:

$$Pr[X \geq (1 - d)const \cdot \ln n] \leq e^{-\frac{d^2 const \cdot \ln n}{2}}.$$

Denote by c_2 the constant $\frac{const \cdot d^2}{2}$. We get:

$$Pr[X \geq (1 - d)const \cdot \ln n] \leq e^{-c_2 \ln n}.$$

Thus,

$$Pr[X < (1-d)const \cdot \ln n] = Pr[X < (1-d)\lambda] > 1 - \frac{1}{e^{c_2 \ln n}} = 1 - \frac{1}{n^{c_2}}.$$

◀

▶ **Lemma 30** (S3). *At least W processes in $C(s,\lambda)$ are correct whp.*

Proof. Let X be a random variable that represents the number of correct processes that are sampled to $C(s,\lambda)$. $X \sim Bin((\frac{2}{3}+\epsilon)n, \frac{const \cdot \ln n}{n})$, thus $E[X] = (\frac{2}{3}+\epsilon)const \cdot \ln n$. Let $d' = 3d + \frac{1}{\lambda}$. Notice that $1 - \frac{\frac{2}{3}+d'}{\frac{2}{3}+\epsilon} \leq 1$ and also $1 - \frac{\frac{2}{3}+d'}{\frac{2}{3}+\epsilon} = 1 - \frac{\frac{2}{3}+3d+\frac{1}{\lambda}}{\frac{2}{3}+\epsilon} = \frac{\frac{2}{3}+\epsilon-\frac{2}{3}-3d-\frac{1}{\lambda}}{\frac{2}{3}+\epsilon} \geq \frac{\frac{3}{\lambda}+\frac{1}{\lambda}-3d-\frac{1}{\lambda}}{\frac{2}{3}+\epsilon} \geq 0$. Hence, we can put $\delta = 1 - \frac{\frac{2}{3}+d'}{\frac{2}{3}+\epsilon}$ in (4) and get:

$$Pr[X \leq (1-(1-\frac{\frac{2}{3}+d'}{\frac{2}{3}+\epsilon}))(\frac{2}{3}+\epsilon)const \cdot \ln n] \leq e^{-\frac{(1-\frac{\frac{2}{3}+d'}{\frac{2}{3}+\epsilon})^2(\frac{2}{3}+\epsilon)const \cdot \ln n}{2}},$$

$$Pr[X \leq (\frac{\frac{2}{3}+d'}{\frac{2}{3}+\epsilon})(\frac{2}{3}+\epsilon)const \cdot \ln n] \leq e^{-\frac{(1-\frac{\frac{2}{3}+d'}{\frac{2}{3}+\epsilon})^2(\frac{2}{3}+\epsilon)const \cdot \ln n}{2}},$$

$$Pr[X \leq (\frac{2}{3}+d')const \cdot \ln n] \leq e^{-\frac{(1-\frac{\frac{2}{3}+d'}{\frac{2}{3}+\epsilon})^2(\frac{2}{3}+\epsilon)const \cdot \ln n}{2}}.$$

Denote by c_3 the constant $\frac{const \cdot (1-\frac{\frac{2}{3}+d'}{\frac{2}{3}+\epsilon})^2(\frac{2}{3}+\epsilon)}{2}$. We get:

$$Pr[X \leq (\frac{2}{3}+d')const \cdot \ln n] \leq e^{-c_3 \ln n}.$$

Thus,

$$Pr[X > (\frac{2}{3}+d')const \cdot \ln n] = Pr[X > (\frac{2}{3}+d')\lambda] > 1 - \frac{1}{e^{c_3 \ln n}} = 1 - \frac{1}{n^{c_3}}.$$

To this point we've proved that at least $(\frac{2}{3}+d')\lambda$ processes in $C(s,\lambda)$ are correct whp. It follows that at least $(\frac{2}{3}+3d+\frac{1}{\lambda})\lambda = (\frac{2}{3}+3d)\lambda + 1$ processes in $C(s,\lambda)$ are correct whp.

As $\lceil(\frac{2}{3}+3d)\lambda\rceil \leq (\frac{2}{3}+3d)\lambda + 1$ we conclude that at least $W = \lceil(\frac{2}{3}+3d)\lambda\rceil$ processes in $C(s,\lambda)$ are correct whp.

◀

▶ **Lemma 31** (S4). *At most B processes in $C(s,\lambda)$ are Byzantine whp.*

Proof. Let X be a random variable that represents the number of Byzantine processes that are sampled to $C(s,\lambda)$. $X \sim Bin((\frac{1}{3}-\epsilon)n, \frac{const \cdot \ln n}{n})$, thus $E[X] = (\frac{1}{3}-\epsilon)const \cdot \ln n$.

Placing $\delta = \frac{\epsilon-d}{\frac{1}{3}-\epsilon} \geq 0$ in (3) we get:

$$Pr[X \geq (1+\frac{\epsilon-d}{\frac{1}{3}-\epsilon})(\frac{1}{3}-\epsilon)const \cdot \ln n] \leq e^{-\frac{(\frac{\epsilon-d}{\frac{1}{3}-\epsilon})^2(\frac{1}{3}-\epsilon)const \cdot \ln n}{2+(\frac{\epsilon-d}{\frac{1}{3}-\epsilon})}},$$

$$Pr[X \geq (\frac{\frac{1}{3}-d}{\frac{1}{3}-\epsilon})(\frac{1}{3}-\epsilon)const \cdot \ln n] \leq e^{-\frac{\frac{(\epsilon-d)^2}{\frac{1}{3}-\epsilon}const\cdot\ln n}{2+(\frac{\epsilon-d}{\frac{1}{3}-\epsilon})}},$$

$$Pr[X \geq (\frac{1}{3}-d)const \cdot \ln n] \leq e^{-\frac{\frac{(\epsilon-d)^2}{\frac{1}{3}-\epsilon}const\cdot\ln n}{2+(\frac{\epsilon-d}{\frac{1}{3}-\epsilon})}}.$$

Denote by c_4 the constant $\frac{const\cdot\frac{(\epsilon-d)^2}{\frac{1}{3}-\epsilon}}{2+(\frac{\epsilon-d}{\frac{1}{3}-\epsilon})}$. We get:

$$Pr[X \geq (\frac{1}{3}-d)const \cdot \ln n] \leq e^{-c_4 \ln n}.$$

Thus,

$$Pr[X < (\frac{1}{3}-d)const \cdot \ln n] = Pr[X < (\frac{1}{3}-d)\lambda] > 1 - \frac{1}{e^{c_4 \ln n}} = 1 - \frac{1}{n^{c_4}}.$$

Since X must be an integer, it follows that $X \leq B = \lfloor(\frac{1}{3}-d)\lambda\rfloor$ whp.

◀

◀

▶ **Corollary 32** (S5). *Consider $C(s,\lambda)$ for some string s and some $\lambda = const \cdot \ln n$ and two sets $P_1, P_2 \subset C(s,\lambda)$ s.t $|P_1| = |P_2| = W$. Then, $|P_1 \cap P_2| \geq B+1$.*

Proof. The set P_2 contains at most $|C(s,\lambda) \setminus P_1|$ processes that aren't in P_1. By S1, and since $P_1 \subset C(s,\lambda)$:

$$|C(s,\lambda)\setminus P_1| \leq (1+d)\lambda - W = (1+d)\lambda - \left\lceil(\frac{2}{3}+3d)\lambda\right\rceil \leq (1+d)\lambda - (\frac{2}{3}+3d)\lambda = (\frac{1}{3}-2d)\lambda.$$

The remaining processes in P_2 are also in P_1, so

$$|P_1 \cap P_2| \geq W - (\frac{1}{3}-2d)\lambda = \left\lceil(\frac{2}{3}+3d)\lambda\right\rceil - (\frac{1}{3}-2d)\lambda \geq (\frac{2}{3}+3d)\lambda - (\frac{1}{3}-2d)\lambda = (\frac{1}{3}+5d)\lambda.$$

Finally,

$$|P_1 \cap P_2| - B = |P_1 \cap P_2| - \left\lfloor(\frac{1}{3}-d)\lambda\right\rfloor \geq (\frac{1}{3}+5d)\lambda - (\frac{1}{3}-d)\lambda = 6d\lambda > \frac{6\lambda}{\lambda} \geq 1,$$

as requested. ◀

▶ **Corollary 33** (S6). *Consider $C(s,\lambda)$ for some string s and some $\lambda = const \cdot \ln n$ and two sets $P_1, P_2 \subset C(s,\lambda)$ s.t $|P_1| = B+1$ and $|P_2| = W$. Then, $|P_1 \cap P_2| \geq 1$.*

Proof. The set P_2 contains at most $|C(s,\lambda) \setminus P_1|$ processes that aren't in P_1. By S1, and since $P_1 \subset C(s,\lambda)$:

$$|C(s,\lambda)\setminus P_1| \leq (1+d)\lambda - (B+1) = (1+d)\lambda - (\left\lfloor(\frac{1}{3}-d)\lambda\right\rfloor + 1) \leq (1+d)\lambda - ((\frac{1}{3}-d)\lambda - 1) - 1 = (\frac{2}{3}+2d)\lambda.$$

Therefore,

$$|P_2| - |C(s,\lambda)\setminus P_1| \geq W - (\frac{2}{3}+2d)\lambda = \left\lceil(\frac{2}{3}+3d)\lambda\right\rceil - (\frac{2}{3}+2d)\lambda \geq (\frac{2}{3}+3d)\lambda - (\frac{2}{3}+2d)\lambda = d\lambda > \frac{\lambda}{\lambda} = 1,$$

and so $|P_1 \cap P_2| \geq 1$, as requested. ◀

Appendix B WHP Coin Proofs

In the committee-based protocol, a value v is *common* if at least $B+1$ correct processes in $C(\text{SECOND}, \lambda)$ have $v_i = v$ at the end of phase 1. The next lemma adapts the lower bound of Lemma 3 on the number of common values to the committee-based protocol.

▶ **Lemma 34.** *In Algorithm 2 whp,* $c \geq \frac{d(11-3d)}{1+9d}\lambda$.

Proof. Let $n_1 = |C(\text{FIRST}, \lambda)|, n_2 = |C(\text{SECOND}, \lambda)|$. We define a table T with n_2 rows and n_1 columns. For each correct process $p_i \in C(\text{SECOND}, \lambda)$ and each $0 \leq j \leq n_1 - 1$, $T[i,j] = 1$ iff p_i receives $\langle \text{FIRST}, v \rangle$ from $p_j \in P_1$ before sending the SECOND message in line 10. Each row of a correct process contains exactly W ones since it waits for W $\langle \text{FIRST}, v \rangle$ messages (line 9). Each row of a faulty process in $C(\text{SECOND}, \lambda)$ is arbitrarily filled with W ones and $n_1 - W$ zeros. Thus the total number of ones in the table is $n_2 W$ and the total number of zeros is $n_2(n_1 - W)$. Let k be the number of columns with at least $2B+1$ ones. Each column represents a value sent by a process in $C(\text{FIRST}, \lambda)$. By S4, whp, at most B of the processes that receive this value are Byzantine. Thus, whp, out of any $2B+1$ ones in each of these columns, at least $B+1$ represent correct processes that receive this value and it follows that $c \geq k$.

Denote by x the number of ones in the remaining columns. Because each column has at most n_2 ones we get:

$$x \geq n_2 W - k n_2 = n_2 \left\lceil (\frac{2}{3} + 3d)\lambda \right\rceil - k n_2 \geq n_2 (\frac{2}{3} + 3d)\lambda - k n_2. \tag{5}$$

And because the remaining columns have at most $2B$ ones:

$$x \leq 2B(n_1 - k) = 2 \left\lfloor (\frac{1}{3} - d)\lambda \right\rfloor (n_1 - k) \leq 2(\frac{1}{3} - d)\lambda(n_1 - k). \tag{6}$$

Combining (1), (2) we get:

$$2(\frac{1}{3} - d)\lambda(n_1 - k) \geq n_2(\frac{2}{3} + 3d)\lambda - k n_2$$

$$k n_2 - 2\lambda k(\frac{1}{3} - d) \geq n_2(\frac{2}{3} + 3d)\lambda - 2(\frac{1}{3} - d)\lambda n_1$$

$$k(n_2 - 2\lambda(\frac{1}{3} - d)) \geq \lambda(n_2(\frac{2}{3} + 3d) - 2(\frac{1}{3} - d)n_1)$$

$$k \geq \frac{\lambda(n_2(\frac{2}{3} + 3d) - 2(\frac{1}{3} - d)n_1)}{n_2 - 2\lambda(\frac{1}{3} - d)}$$

By S2 for $C(\text{SECOND}, \lambda)$, whp $n_2 \geq (1-d)\lambda$ and we get:

$$k \geq \frac{\lambda((1-d)\lambda(\frac{2}{3} + 3d) - 2(\frac{1}{3} - d)n_1)}{n_2 - 2\lambda(\frac{1}{3} - d)}$$

By S1 for $C(\textsc{first}, \lambda)$ and $C(\textsc{second}, \lambda)$, whp $n_1, n_2 \leq (1+d)\lambda$ and we get:

$$k \geq \frac{\lambda\Big[(1-d)\lambda(\frac{2}{3}+3d) - 2(\frac{1}{3}-d)(1+d)\lambda\Big]}{(1+d)\lambda - 2\lambda(\frac{1}{3}-d)} = \frac{\lambda\Big[(1-d)(\frac{2}{3}+3d) - 2(\frac{1}{3}-d)(1+d)\Big]}{(1+d) - 2(\frac{1}{3}-d)}$$

Finally, we get whp:

$$c \geq k \geq \frac{d(11-3d)}{1+9d}\lambda.$$

as required. ◀

Let $v_{min} \triangleq \min_{p_i \in C(\textsc{first},\lambda)} \{VRF_i(r)\}$. Similiarly to Lemma 4, we prove that the probability that it is common is bounded by a constant, whp. I.e., we show that $Prob[v_{min} \textit{ is common}] \geq const \cdot g(n)$ where $g(n)$ goes to 1 as n goes to infinity.

▶ **Lemma 35.** *whp* $Prob[v_{min} \textit{ is common}] \geq \frac{2}{3(1-d)} \cdot \frac{c-B}{(1+d)\lambda - B}$.

Proof. Notice that we assume that the invocation of whp_coin(r) by every process is causally independent of its progress at other processes. Hence, for any two processes $p_i, p_j \in C(\textsc{first}, \lambda)$, the messages $\langle \textsc{first}, v_i \rangle$, $\langle \textsc{first}, v_j \rangle$ are causally concurrent. Thus, due to our *delayed-adaptive adversary* definition, these messages are scheduled by the adversary regardless of their content, namely their VRF random values. Notice that the adversary can corrupt processes before they initially send their VRF values. By S4 there are at most B Byzantine processes in $C(\textsc{first}, \lambda)$. Since the adversary cannot predict the VRF outputs, the probability for a given process to be corrupted before sending its FIRST messages is at most $\frac{B}{|C(\textsc{first},\lambda)|}$. The adversary is oblivious to the correct processes' VRF values when it schedules their first phase messages. Therefore, each of them has the same probability to become common. Since at most B common values are from Byzantine processes, this probability is at least $\frac{c-B}{|C(\textsc{first},\lambda)|-B}$. We conclude that v_{min} is common with probability at least $(1 - \frac{B}{|C(\textsc{first},\lambda)|})\frac{c-B}{|C(\textsc{first},\lambda)|-B}$. By S1 and S2 we get that $(1-d)\lambda \leq |C(\textsc{first}, \lambda)| \leq (1+d)\lambda$ whp.

Thus, whp, v_{min} is common with probability at least $(1 - \frac{B}{(1-d)\lambda})\frac{c-B}{(1+d)\lambda-B} = (1 - \frac{\lfloor(\frac{1}{3}-d)\lambda\rfloor}{(1-d)\lambda})\frac{c-B}{(1+d)\lambda-B} \geq (1 - \frac{(\frac{1}{3}-d)\lambda}{(1-d)\lambda})\frac{c-B}{(1+d)\lambda-B} = \frac{2}{3(1-d)} \cdot \frac{c-B}{(1+d)\lambda-B}$. ◀

▶ **Lemma 36.** *If v_{min} is common then whp each correct process holds v_{min} at the end of phase 2.*

Proof. Since v_{min} is common, at least $B+1$ correct members of $C(\textsc{second}, \lambda)$ receive it by the end of phase 1 and update their local values to v_{min}. During the second phase, each correct process hears from W members of $C(\textsc{second}, \lambda)$ whp. By S6, this means that it hears from at least one correct process that has updated its value to v_{min} and sent it whp. ◀

▶ **Lemma 37.** *Let* $\rho = \frac{18d^2+27d-1}{3(5+6d)(1-d)(1+9d)}$. *Algorithm 2 implements a shared coin with success rate ρ, whp.*

Proof. Denote $n_1 = |C(\textsc{first}, \lambda)|$. We bound whp the probability that all correct processes output $b \in \{0,1\}$ as follows:

$Prob$[all correct processes output b] $\geq Prob$[all correct processes have the same v_i at the end of phase 2 and its LSB is b] $\geq Prob$[all correct processes have $v_i = v_{min}$ at the end of phase 2 and its LSB is b] $= \frac{1}{2} \cdot Prob$[all correct processes have $v_i = v_{min}$] $\overset{\text{Lemma 36}}{\geq} \frac{1}{2} \cdot Prob[v_{min}$ is common] $\overset{\text{Lemma 35}}{\geq} \frac{1}{2} \cdot \frac{2}{3(1-d)} \cdot \frac{c-B}{(1+d)\lambda - B} \overset{\text{Lemma 34}}{\geq} \frac{1}{3(1-d)} \cdot \frac{\frac{d(11-3d)}{1+9d}\lambda - B}{(1+d)\lambda - B} = \frac{1}{3(1-d)} \cdot \frac{\frac{d(11-3d)}{1+9d}\lambda - \lfloor(\frac{1}{3}-d)\lambda\rfloor}{(1+d)\lambda - \lfloor(\frac{1}{3}-d)\lambda\rfloor} \geq \frac{1}{3(1-d)} \cdot \frac{\frac{d(11-3d)}{1+9d}\lambda - (\frac{1}{3}-d)\lambda}{(1+d)\lambda - ((\frac{1}{3}-d)\lambda - 1)} = \frac{1}{3(1-d)} \cdot \frac{\lambda\frac{18d^2+27d-1}{27d+3}}{\lambda(\frac{2}{3}+2d)+1} \geq \frac{1}{3(1-d)} \cdot \frac{\lambda\frac{18d^2+27d-1}{27d+3}}{\lambda(\frac{2}{3}+2d)+\lambda} = \frac{18d^2+27d-1}{3(5+6d)(1-d)(1+9d)}$.

◀

We have shown a bound on the coin's success rate whp. Since $d > 0.0362$, the coin's success rate is a positive constant whp. We next prove that the coin ensures liveness whp.

▶ **Lemma 38.** *If all correct processes invoke Algorithm 2 then all correct processes return whp.*

Proof. All correct processes in $C(\textsc{first}, \lambda)$ send their message in the first phase. At least W of them are correct whp by S3. All correct processes in $C(\textsc{second}, \lambda)$ eventually receive W $\langle \textsc{first}, x \rangle$ messages whp and send a message in the second phase. As ,whp, again W correct processes send their messages (by S3), each correct process eventually receives W $\langle \textsc{second}, x \rangle$ messages and returns whp. ◀

From Lemma 37 and Lemma 38 we conclude:

▶ **Theorem 14.** *Algorithm 2 implements a WHP coin with a constant success rate.*

2.2 Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer

Appears in the 35th International Symposium on Distributed Computing (DISC 2021).

Tame the Wild with Byzantine Linearizability: Reliable Broadcast, Snapshots, and Asset Transfer

Shir Cohen
Technion, Israel

Idit Keidar
Technion, Israel

Abstract

We formalize Byzantine linearizability, a correctness condition that specifies whether a concurrent object with a sequential specification is resilient against Byzantine failures. Using this definition, we systematically study Byzantine-tolerant emulations of various objects from registers. We focus on three useful objects– reliable broadcast, atomic snapshot, and asset transfer. We prove that there exist n-process f-resilient Byzantine linearizable implementations of such objects from registers if and only if $f < \frac{n}{2}$.

2012 ACM Subject Classification Theory of computation → Concurrent algorithms

Keywords and phrases Byzantine linearizability, concurrent algorithms, snapshot, asset transfer

1 Introduction

Over the last decade, cryptocurrencies have taken the world by storm. The idea of a decentralized bank, independent of personal motives has gained momentum, and cryptocurrencies like Bitcoin [23], Ethereum [25], and Diem [8] now play a big part in the world's economy. At the core of most of these currencies lies the asset transfer problem. In this problem, there are multiple accounts, operated by processes that wish to transfer assets between accounts. This environment raises the need to tolerate the malicious behavior of processes that wish to sabotage the system.

In this work, we consider the shared memory model that was somewhat neglected in the Byzantine discussion. We believe that shared memory abstractions, implemented in distributed settings, allow for an intuitive formulation of the services offered by blockchains and similar decentralized tools. It is well-known that it is possible to implement reliable read-write shared memory registers via message passing even if a fraction of the servers are Byzantine [1, 21, 24, 19]. As a result, as long as the client processes using the service are not malicious, any fault-tolerant object that can be constructed using registers can also be implemented in the presence of Byzantine servers. However, it is not clear what can be done with such objects when they are used by Byzantine client processes. In this work, we study this question.

In Section 4 we define *Byzantine linearizability*, a correctness condition applicable to any shared memory object with a sequential specification. Byzantine linearizability addresses the usage of reliable shared memory abstractions by potentially Byzantine client processes. We then systematically study the feasibility of implementing various Byzantine linearizable shared memory objects from registers.

We observe that existing Byzantine fault-tolerant shared memory constructions [20, 22, 1] in fact implement Byzantine linearizable registers. Such registers are the starting point of our study. When trying to implement more complex objects (e.g., snapshots and asset transfer) using registers, constructions that work in the crash-failure model no longer work when Byzantine processes are involved, and new algorithms – or impossibility results – are needed.

As our first result, we prove in Section 5 that an asset transfer object used by Byzantine client processes does not have a wait-free implementation, even when its API is reduced to support only transfer operations (without reading processes' balances). Furthermore, it cannot be implemented without a majority of correct processes constantly taking steps. Asset transfer has wait-free implementations from both reliable broadcast [7] and snapshots [17] (which we adapt to a Byzantine version) and thus the same lower bound applies to reliable broadcast and snapshots as well.

In Section 6, we present a Byzantine linearizable reliable broadcast algorithm with resilience $f < \frac{n}{2}$, proving that, for this object, the resilience bound is tight. To do so, we define a sequential specification of a reliable broadcast object. Briefly, the object exposes broadcast and deliver operations and we require that deliver return messages previously broadcast. We show that a Byzantine linearizable implementation of such an object satisfies the classical (message-passing) definition [10]. Finally, in Section 7 we present a Byzantine linearizable snapshot with the same resilience. In contrast, previous constructions of Byzantine lattice agreement, which can be directly constructed from a snapshot [6], required $3f + 1$ processes to tolerate f failures.

All in all, we establish a tight bound on the resilience of emulations of three useful shared memory objects from registers. On the one hand, we show that it is impossible to obtain wait-free solutions as in the non-Byzantine model, and on the other hand, unlike previous snapshot and lattice agreement algorithms, our solutions do not require $n > 3f$. Taken jointly, our results yield the following theorem:

▶ **Theorem 1.** *In the Byzantine shared memory model, there exist n-process f-resilient Byzantine linearizable implementations of reliable broadcast, snapshot, and asset transfer objects from registers if and only if $f < \frac{n}{2}$.*

Although the construction of reliable registers in message passing systems requires $n > 3f$ servers, our improved resilience applies to client processes, which are normally less reliable than servers, particularly in the so-called *permissioned model* where servers are trusted and clients are ephemeral.

In summary, we make the following contributions:

- Formalizing Byzantine linearizability for any object with a sequential specification.
- Proving that some of the most useful building blocks in distributed computing, such as atomic snapshot and reliable broadcast, do not have f-resilient implementations from SWMR registers when $f \geq \frac{n}{2}$ processes are Byzantine.
- Presenting Byzantine linearizable implementations of a reliable broadcast object and a snapshot object with the optimal resilience.

2 Related Work

In [4] Aguilera et al. present a non-equivocating broadcast algorithm in shared memory. This broadcast primitive is weaker than reliable broadcast – it does not guarantee that all correct processes deliver the same messages, but rather that they do not deliver conflicting messages. A newer version of their work [5], developed concurrently and independently of our work[1], also implements reliable broadcast with $n \geq 2f + 1$, which is very similar to our implementation. While the focus of their work is in the context of RDMA in the *M&M* (message–and–memory) model, our work focuses on the classical shared memory model, which can be emulated in classical message passing systems. While the algorithms are similar,

[1] Their work [5] was in fact published shortly after the initial publication of our results [14].

we formulate reliable broadcast as a shared memory object, with designated API method signatures, which allows us to reason about the operation interval as needed for proving (Byzantine) linearizability and for using this object in constructions of other shared memory objects.

Given a reliable broadcast object, there are known implementations of lattice agreement [16, 26], which resembles a snapshot object. However, these constructions require $n = 3f + 1$ processes. In our work, we present both Byzantine linearizable reliable broadcast and Byzantine snapshot, (from which Byzantine lattice agreement can be constructed [6]), with resilience $n = 2f + 1$.

The asset transfer object we discuss in this paper was introduced by Guerraoui et al. [17, 15]. Their work provides a formalization of the cryptocurrency definition [23]. The highlight of their work is the observation that the asset transfer problem can be solved without consensus. It is enough to maintain a partial order of transactions in the systems, and in particular, every process can record its own transactions. They present a wait-free linearizable implementation of asset transfer in crash-failure shared memory, taking advantage of an atomic snapshot object. We show that we can use their solution, together with our Byzantine snapshot, to solve Byzantine linearizable asset transfer with $n = 2f + 1$.

In addition, Guerraoui et al. present a Byzantine-tolerant solution in the message passing model. This algorithm utilizes reliable broadcast, where dependencies of transactions are explicitly broadcast along with the transactions. This solution does not translate to a Byzantine linearizable one, but rather to a sequentially consistent asset transfer object. In particular, reads can return old (superseded) values, and transfers may fail due to outdated balance reads.

Finally, recent work by Auvolat et al. [7] continues this line of work. They show that a FIFO order property between each pair of processes is sufficient in order to solve the asset transfer problem. This is because transfer operations can be executed once a process's balance becomes sufficient to perform a transaction and there is no need to wait for all causally preceding transactions. However, as a result, their algorithm is not sequentially consistent, or even causally consistent for that matter. For example, assume process i maintains an invariant that its balance is always at least 10, and performs a transfer with amount 5 after another process deposits 5 into its account, increasing its balance to 15. Using the protocol in [7], another process might observe i's balance as 5 if it sees i's outgoing transfer before the causally preceding deposit. Because our solution is Byzantine linearizable, such anomalies are prevented.

3 Model and Preliminaries

We study a distributed system in the shared memory model. Our system consists of a well-known static set $\Pi = \{1, \dots, n\}$ of asynchronous client processes. These processes have access to some shared memory objects. In the shared memory model, all communication between processes is done through the API exposed by the objects in the system: processes invoke operations that in turn, return some response to the process. In this work, we assume a reliable shared memory. (Previous works have presented constructions of such reliable shared memory in the message passing model [1, 21, 24, 3, 19]). We further assume an adversary that may adaptively corrupt up to f processes in the course of a run. When the adversary corrupts a process, it is defined as *Byzantine* and may deviate arbitrarily from the protocol. As long as a process is not corrupted by the adversary, it is *correct*, follows the protocol, and takes infinitely many steps. In particular, it continues to invoke the object's

API infinitely often. Later in the paper, we show that the latter assumption is necessary.

We enrich the model with a *public key infrastructure* (PKI). That is, every process is equipped with a public-private key pair used to sign data and verify signatures of other processes. We denote a value v signed by process i as $\langle v \rangle_i$.

Executions and Histories. We discuss algorithms emulating some object O from lower level objects (e.g., registers). An algorithm is organized as methods of O. A method execution is a sequence of *steps*, beginning with the method's invocation (invoke step), proceeding through steps that access lower level objects (e.g., register read/write), and ending with a return step. The invocation and response delineate the method's execution interval. In an *execution* σ of a Byzantine shared memory algorithm, each correct process invokes methods sequentially, where steps of different processes are interleaved. Byzantine processes take arbitrary steps regardless of the protocol. The *history* H of an execution σ is the sequence of high-level invocation and response events of the emulated object O in σ.

A *sub-history* of a history H is a sub-sequence of the events of H. A history H is *sequential* if it begins with an invocation and each invocation, except possibly the last, is immediately followed by a matching response. Operation op is pending in a history H if op is invoked in H but does not have a matching response event.

A history defines a partial order on operations: operation op_1 precedes op_2 in history H, denoted $op_1 \prec_H op_2$, if the response event of op_1 precedes the invocation event of op_2 in H. Two operations are concurrent if neither precedes the other.

Linearizability. A popular correctness condition for concurrent objects in the crash-fault model is linearizability [18], which is defined with respect to an object's sequential specification. A *linearization* of a concurrent history H of object o is a sequential history H' such that (1) after removing some pending operations from H and completing others by adding matching responses, it contains the same invocations and responses as H', (2) H' preserves the partial order $\prec_H$, and (3) H' satisfies o's sequential specification.

f-resilient. An algorithm is *f-resilient* if as long as at most f processes fail, every correct process eventually returns from each operation it invokes. A *wait-free* algorithm is a special case where $f = n - 1$.

Single Writer Multiple Readers Register. The basic building block in shared memory is a single writer multiple readers (SWMR) register that exposes *read* and *write* operations. Such registers are used to construct more complicated objects. The sequential specification of a SWMR register states that every read operation from register R returns the value last written to R. Note that if the writer is Byzantine, it can cause a correct reader to read arbitrary values.

Asset Transfer Object. In [17, 15], the asset transfer problem is formulated as a sequential object type, called *Asset Transfer Object.* The asset transfer object maintains a mapping from processes in the system to their balances[2]. Initially, the mapping contains the initial balances of all processes. The object exposes a *transfer* operation, *transfer(src,dst,amount)*, which can be invoked by process *src* (only). It withdraws *amount* from process *src*'s account and deposits it at process *dst*'s account provided that *src*'s balance was at least *amount.* It returns a boolean that states whether the transfer was successful (i.e., *src* had *amount* to spend). In addition, the object exposes a *read(i)* operation that returns the current balance of i.

[2] The definition in [17] allows processes to own multiple accounts. For simplicity, we assume a single account per-process, as in [15].

4 Byzantine Linearizability

In this section we define Byzantine linearizability. Intuitively, we would like to tame the Byzantine behavior in a way that provides consistency to correct processes. We linearize the correct processes' operations and offer a degree of freedom to embed additional operations by Byzantine processes.

We denote by $H|_{correct}$ the projection of a history H to all correct processes. We say that a history H is Byzantine linearizable if $H|_{correct}$ can be augmented with operations of Byzantine processes such that the completed history is linearizable. That is, there is another history, with the same operations by correct processes as in H, and additional operations by another at most f processes. In particular, if there are no Byzantine failures then Byzantine linearizability is simply linearizability. Formally:

▶ **Definition 2.** *(Byzantine Linearizability) A history H is Byzantine linearizable if there exists a history H' so that $H'|_{correct} = H|_{correct}$ and H' is linearizable.*

Similarly to linearizability, we say that an object is Byzantine linearizable if all of its executions are Byzantine Linearizable.

Next, we characterize objects for which Byzantine linearizability is meaningful. The most fundamental component in shared memory is read-write registers. Not surprisingly, such registers, whether they are single-writer or multi-writers ones are de facto Byzantine linearizable without any changes. This is because before every read from a Byzantine register, invoked by a correct process, one can add a corresponding Byzantine write.

In practice, multiple writers multiple readers (MWMR) registers are useless in a Byzantine environment as an adversary that controls the scheduler can prevent any communication between correct processes. SWMR registers, however, are still useful for constructing more meaningful objects. Nevertheless, the constructions used in the crash-failure model for linearizable objects do not preserve this property. For instance, if we allow Byzantine processes to run a classic atomic snapshot algorithm [2] using Byzantine linearizable SWMR registers, it will not result in a Byzantine linearizable snapshot object. The reason is that the algorithm relies on correct processes being able to perform "double-collect" meaning that at some point a correct process manages to read all registers twice without witnessing any changes. While this is true in the crash-failure model, in the Byzantine model this is not the case as the adversary can change some registers just before any correct read.

Relationship to Other Correctness Conditions

Byzantine linearizability provides a simple and intuitive way to capture Byzantine behavior in the shared memory model. We now examine the relationship of Byzantine linearizability with previously suggested correctness conditions involving Byzantine processes.

PBFT [12, 11] presented a formalization of linearizability in the presence of Byzantine-faulty clients in message passing systems. Their notion of linearizability is formulated in the form of I/O automata. Their specification is in the same spirit as ours, but our formulation is closer to the original notion of linearizability in shared memory.

Some works have defined linearization conditions for specific objects. This includes conditions for SWMR registers [22], a distributed ledger [13], and asset transfer [7]. Our condition coincides with these definitions for the specific objects and thus generalizes all of them. Liskov and Rodrigues [20] presented a correctness condition that has additional restrictions. Their correctness notion relies on the idea that Byzantine processes are eventually detected and removed from the system and focuses on converging to correct system behavior after their departure. While this model is a good fit when the threat model is software bugs or malicious intrusions, it is less appropriate for settings like cryptocurrencies, where Byzantine behavior cannot be expected to eventually stop.

5 Lower Bound on Resilience

In shared memory, one typically aims for wait-free objects, which tolerate any number of process failures. Indeed, many useful objects have wait-free implementations from SWMR registers in the non-Byzantine case. This includes reliable broadcast, snapshots, and as recently shown, also asset transfer. We now show that in the Byzantine case, wait-free implementations of these objects are impossible. Moreover, a majority of correct processes is required.

▶ **Theorem 3.** *In the Byzantine shared memory model, for any $f > 2$, there does not exist a Byzantine linearizable implementation of asset transfer that supports only transfer operations in a system with $n \leq 2f$ processes, f of which can be Byzantine, using only SWMR registers.*

Note that to prove this impossibility, it does not suffice to introduce bogus actions by Byzantine processes, because the notion of Byzantine linearizability allows us to ignore these actions. Rather, to derive the contradiction, we create runs where the bogus behavior of the Byzantine processes leads to incorrect behavior of the correct processes.

Proof. Assume by contradiction that there is such an algorithm. Let us look at a system with $n = 2f$ correct processes. Partition Π as follows: $\Pi = A \cup B \cup \{p_1, p_2\}$, where $|A| = f - 1$, $|B| = f - 1$, $A \cap B = \emptyset$, and $p_1, p_2 \notin A \cup B$. By assumption, $|A| > 1$. Let z be a process in A. Also, by assumption $|B| \geq 2$. Let q_1, q_2 be processes in B. The initial balance of all processes but z is 0, and the initial balance of z is 1. We construct four executions as shown in Figure 1.

Let σ_1 be an execution where, only processes in $A \cup \{p_1\}$ take steps. First, z performs *transfer(z, p_1,1)*. Since up to f processes may be faulty, the operation completes, and by the object's sequential specification, it is successful (returns true). Then, p_1 performs *transfer(p_1, q_1,1)*. By f-resilience and linearizability, this operation also completes successfully. Note that in σ_1 no process is actually faulty, but because of f-resilience, progress is achieved when f processes are silent.

Similarly, let σ_2 be an execution where the processes in $A \cup \{p_2\}$ are correct, and z performs *transfer(z, p_2,1)*, followed by p_2 performing *transfer(p_2, q_2,1)*.

We now construct σ_3, where all processes in $A \cup \{p_1\}$ are Byzantine. We first run σ_1. Call the time when it ends t_1. At this point, all processes in $A \cup \{p_1\}$ restore their registers to their initial states. Note that no other processes took steps during σ_1, hence the entire shared memory is now in its initial state. Then, we execute σ_2. Because we have reset the memory to its initial state, the operations execute the same way. When σ_2 completes, processes in $A \backslash \{z\} \cup \{p_1\}$ restore their registers to their state at time t_1. At this point, the state of z and p_2 is the same as it was at the end of σ_2, the state of processes in $A \setminus \{z\} \cup \{p_1\}$ is the same as it was at the end of σ_1, and processes in B are all in their initial states.

We construct σ_4 where all processes in $A \cup \{p_2\}$ are Byzantine by executing σ_2, having all processes in $A \cup \{p_2\}$ reset their memory, executing σ_1, and then having z and p_2 restore their registers to their state at the end of σ_2. At this point, the state of z and p_2 is the same as it was at the end of σ_2, the state of processes in $A \setminus \{z\} \cup \{p_1\}$ is the same as it was at the end of σ_1, and processes in B are all in their initial states.

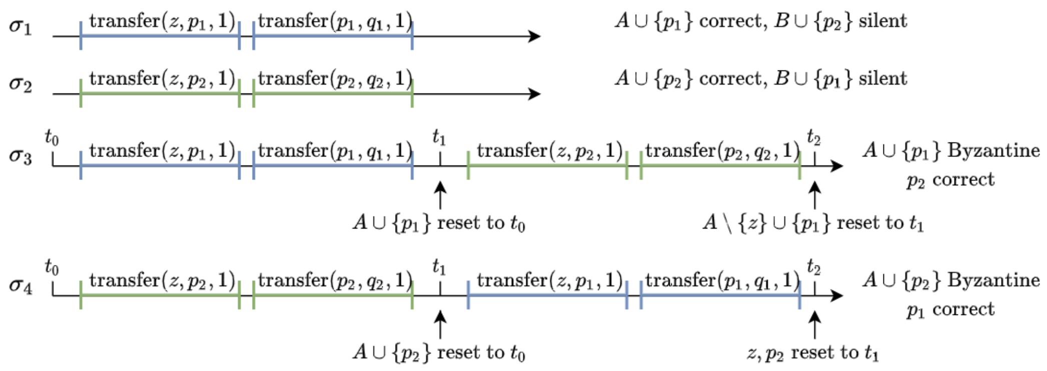

Figure 1 An asset transfer object does not have an f-resilient implementation for $n \leq 2f$.

We observe that for processes in B, the configurations at the end of σ_3 and σ_4 are indistinguishable as they did not take any steps and the global memory is the same. By f-resilience, in both cases q_1 and q_2, together with processes in B and one of $\{p_1, p_2\}$ should be able to make progress at the end of each of these runs. We extend the runs by having q_1 and q_2 invoke transfers of amount 1 to each other. In both runs processes in $B \cup \{p_1, p_2\}$ help them make progress. In σ_3, p_1 behaves as if it is a correct process and its local state is the same as it is at the end of σ_1, and in σ_4 p_2 behaves as if it is a correct process and its local state is the same as it is at the end of σ_2. Thus, σ_3 and σ_4 are indistinguishable to all correct processes, and as a result q_1 and q_2 act the same in both runs. However, from safety exactly one of their transfers should succeed. In σ_3, p_2 is correct and *transfer(p_2, q_2,1)* succeeds, allowing q_2 to transfer 1 and disallowing the transfer from q_1, whereas σ_4 the opposite is true. This is a contradiction. ◀

Guerraoui et al. [17] use an atomic snapshot to implement an asset transfer object in the crash-fault shared memory model. In addition, they handle Byzantine processes in the message passing model by taking advantage of reliable broadcast. In Appendix A we show that their atomic snapshot-based asset transfer can be easily adapted to the Byzantine settings by using a Byzantine linearizable snapshot, resulting in a Byzantine linearizable asset transfer. Their reliable broadcast-based algorithm, on the other hand, is not linearizable and therefore not Byzantine linearizable even when using Byzantine linearizable reliable broadcast. Nonetheless, Auvolat et al. [7] have used reliable broadcast to construct an asset transfer object where transfer operations are linearizable (although reads are not).

We note that our lower bound holds for an asset transfer object without read operations. This discussion and the construction in Appendix A lead us to the following corollary:

▶ **Corollary 4.** *In the Byzantine shared memory model, for any $f > 2$, there does not exist an f-resilient Byzantine linearizable implementation of an atomic snapshot or reliable broadcast in a system with $f \geq \frac{n}{2}$ Byzantine processes using only SWMR registers.*

Furthermore, we prove in the following lemma that in order to provide f-resilience it is required that at least a majority of correct processes take steps infinitely often, justifying our model definition.

▶ **Lemma 5.** *In the Byzantine shared memory model, for any $f > 2$, there does not exist an f-resilient Byzantine linearizable implementation of asset transfer in a system with $n \geq 2f+1$*

processes, f of which can be Byzantine, using only SWMR registers if less than $f+1$ correct processes take steps infinitely often.

Proof. Assume by way of contradiction that there exists an f-resilient Byzantine linearizable implementation of asset transfer in a system with $n \geq 2f+1$ processes where there are at most f correct processes that take steps infinitely often. Denote these f correct processes by the set A. Thus, there is a point t in any execution such that from time t, only processes in A and Byzantine processes take any steps. Starting t, the implementation is equivalent to one in a system with $n = 2f$, f of them may be Byzantine. This is a contradiction to Theorem 3. ◀

6 Byzantine Linearizable Reliable Broadcast

With the acknowledgment that not all is possible, we seek to find Byzantine linearizable objects that are useful even without a wait-free implementation. One of the practical objects is a reliable broadcast object. We already proved in the previous section that it does not have an f-resilient Byzantine linearizable implementation, for any $f \geq max\{3, \frac{n}{2}\}$. In this section we provide an implementation that tolerates $f < \frac{n}{2}$ faults.

6.1 Reliable Broadcast Object

The reliable broadcast primitive exposes two operations *broadcast(ts,m)* returning void and *deliver(j,ts)* returning m. When $deliver_j(i, ts)$ returns m we say that process j delivers m from process i in timestamp ts. The broadcast operation allows processes to spread a message m in the system, along with some timestamp ts. The use of timestamps allows processes to broadcast multiple messages.

Its classical definition, given for message passing systems [10], requires the following properties:

- Validity: If a correct process i broadcasts (ts, m) then all correct processes eventually deliver m from process i in timestamp ts.
- Agreement: If a correct process delivers m from process i in timestamp ts, then all correct processes eventually deliver m from process i in timestamp ts.
- Integrity: No process delivers two different messages for the same (ts, j) and if j is correct delivers only messages j previously broadcast.

In the shared memory model, the deliver operation for some process j and timestamp ts returns the message with timestamp ts previously broadcast by j, if exists. We define the sequential specification of reliable broadcast as follows:

▶ **Definition 6.** *A reliable broadcast object exposes two operations* broadcast(ts,m) *returning void and* deliver(j,ts) *returning m. A call to* deliver(j,ts) *returns the value m of the first* broadcast(ts,m) *invoked by process j before the deliver operation. If j did not invoke* broadcast *before the deliver, then it returns* $\perp$*.*

Note that as the definition above refers to sequential histories, the first broadcast operation (if such exists) is well-defined. Further, whereas in message passing systems reliable broadcast works in a push fashion, where the receipt of a message triggers action at its destination, in the shared memory model processes need to actively pull information from the registers. A process pulls from another process j using the *deliver(j,ts)* operation and returns with a value $m \neq \perp$. If all messages are eventually pulled, the reliable broadcast properties are achieved, as proven in the following lemma.

▶ **Lemma 7.** *A Byzantine linearization of a reliable broadcast object satisfies the three properties of reliable broadcast.*

Proof. If a correct process broadcasts m, and all messages are subsequently pulled then according to Definition 6 all correct processes deliver m, providing validity. For agreement, if a correct process invokes *deliver(j,ts)* that returns m and all messages are later pulled by all correct processes, it follows that all correct processes also invoke *deliver(j,ts)* and eventually return $m' \neq \perp$. Since *deliver(j,ts)* returns the value v of the first *broadcast(ts,v)* invoked by process j before it is called, and there is only one first broadcast, and we get that $m = m'$. Lastly, if *deliver(j,ts)* returns m, by the specification, j previously invoked *broadcast(ts,m)*. ◀

6.2 Reliable Broadcast Algorithm

In our implementation (given in Algorithm 1), each process has 4 SWMR registers: send, echo, ready, and deliver, to which we refer as *stages* of the broadcast. We follow concepts from Bracha's implementation in the message passing model [9] but leverage the shared memory to improve its resilience from $3f + 1$ to $2f + 1$. The basic idea is that a process that wishes to broadcast value v writes it in its send register (line 4) and returns only when it reaches the deliver stage. I.e., v appears in the deliver register of at least one correct process. Throughout the run, processes infinitely often call a *refresh* function whose role is to help the progress of the system. When refreshing, processes read all registers and help promote broadcast values through the 4 stages. For a value to be delivered, it has to have been read and signed by $f + 1$ processes at the ready stage. Because each broadcast message is copied to 4 registers of each process, the space complexity is $4n$ per message. Whether this complexity can be improved remains as an open question.

In the refresh function, executed for all processes, at first a process reads the last value written to a send register (line 16). If the value is a signed pair of a message and a timestamp, refresh then copies it to the process's echo register in line 18. In the echo register, the value remains as evidence, preventing conflicting values (sent by Byzantine processes) from being delivered. That is, before promoting a value to the ready or deliver stage, a correct process i performs a "double-collect" of the echo registers (in lines 19,21). Namely, after collecting $f + 1$ signatures on a value in ready registers, meaning that it was previously written in the echo of at least one correct process, i re-reads all echo registers to verify that there does not exist a conflicting value (with the same timestamp and sender). Using this method, concurrent deliver operations "see" each other, and delivery of conflicting values broadcast by a Byzantine process is prevented. Before delivering a value, a process writes it to its deliver register with $f + 1$ signatures (line 22). Once one correct process delivers a value, the following deliver calls can witness the $f + 1$ signatures and copy this value directly from its deliver register (line 11).

We make two assumptions on the correct usage of our algorithm. The first is inherently required as shown in Lemma 5:

▶ **Assumption 1.** *All correct processes infinitely often invoke methods of the reliable broadcast API.*

The second is a straight forward validity assumption:

▶ **Assumption 2.** *Correct processes do not invoke* broadcast(ts,val) *twice with the same ts.*

We now prove our reliable broadcast algorithm's correctness. We first notice:

Algorithm 1 Shared Memory Bracha: code for process i

```
shared SWMR registers: $send_i, echo_i, ready_i, deliver_i$

1: procedure CONFLICTING-ECHO($\langle ts, v\rangle_j$)
2:     return $\exists w \neq v, k \in \Pi$ such that $\langle ts, w\rangle_j \in echo_k$

3: procedure BROADCAST(ts,val)
4:     $send_i \leftarrow \langle ts, val\rangle_i$
5:     repeat
6:         $m \leftarrow$ deliver(i,ts)
7:     until $m \neq \bot$                                  ▷ message is deliverable

8: procedure DELIVER(j,ts)
9:     refresh()
10:    if $\exists k \in \Pi$ and $v$ s.t. $\langle\langle ts, v\rangle_j, \sigma\rangle \in deliver_k$ where $\sigma$ is a set of $f+1$ signatures on
   $\langle ready, \langle ts, v\rangle_j\rangle$ then
11:        $deliver_i \leftarrow deliver_i \cup \{\langle\langle ts, v\rangle_j, \sigma\rangle\}$
12:        return $v$
13:    return $\bot$

14: procedure REFRESH
15:    for $j \in [n]$ do
16:        $m \leftarrow send_j$
17:        if $\nexists ts, val$ s.t. $m = \langle ts, val\rangle_j$ then continue      ▷ $m$ is not a signed pair
18:        $echo_i \leftarrow echo_i \cup \{m\}$
19:        if $\neg$conflicting-echo($m$) then
20:            $ready_i \leftarrow ready_i \cup \{\langle ready, m\rangle_i\}$
21:        if $\exists S \subseteq \Pi$ s.t. $|S| \geq f+1, \forall j \in S, \langle ready, m\rangle_j \in ready_j$ and $\neg$conflicting-echo($m$)
   then
22:            $deliver_i \leftarrow deliver_i \cup \{\langle m, \sigma = \{\langle ready, m\rangle_j | j \in S\}\rangle\}$   ▷ $\sigma$ is the set of $f+1$
   signatures
```

▶ **Observation 8.** *If process i is correct and v appears in $echo_i$ or $ready_i$ it is never deleted.*

▶ **Lemma 9.** *If process i is correct and $\langle\langle ts, v\rangle_i, \sigma\rangle$ appears in $deliver_j$ for any process j then i previously invoked $broadcast(ts, v)$.*

Proof. Since we assume unforgeable signatures, i has previously signed $\langle ts, v\rangle$. By the code, this is only possible if i invoked $broadcast(ts, v)$. ◀

We next prove the following lemma, identifying invariants of Algorithm 1.

▶ **Lemma 10.** *Algorithm 1 satisfies the following invariants:*

I1: If $\langle\langle ts, v\rangle_i, \sigma\rangle$ (where σ is a set of $f+1$ ready signatures) appears in $deliver_j$ for any processes i, j, then $\langle ready, \langle ts, v\rangle_i\rangle_k \in ready_k$ for a correct process k.
I2: If $\langle ready, \langle ts, v\rangle_i\rangle_j \in ready_j$ for a correct process j, then $\langle ts, v\rangle_i \in echo_j$.
I3: If $\langle ready, \langle ts, v\rangle_i\rangle_j$ appears in $ready_j$ and $\langle ready, \langle ts, w\rangle_i\rangle_{j'}$ appears in $ready_{j'}$ for any two correct processes j, j' then $v = w$.

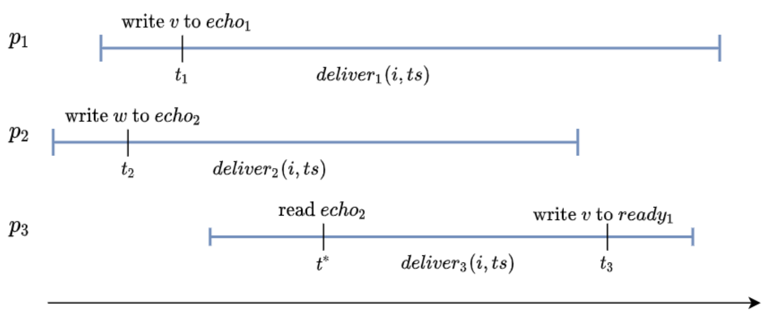

Figure 2 Concurrent deliver operations.

I4: If $\langle\langle ts, v\rangle_i, \sigma\rangle$ appears in $deliver_j$ and $\langle\langle ts, w\rangle_i, \sigma\rangle$ appears in $deliver_{j'}$ for any two correct processes j, j' then $v = w$.

Proof. I1: Since $\langle\langle ts, v\rangle_i, \sigma\rangle$ appears in $deliver_j$ and it contains a set of $f+1$ signatures on $\langle ready, \langle ts, v\rangle_i\rangle$, there is at least one correct process k that signed $\langle ready, \langle ts, v\rangle_i\rangle$ and added it to its ready register. By Observation 8, it is not deleted from the register.

I2: Immediate from the code and Observation 8.

I3: Since $\langle ready, \langle ts, v\rangle_i 0_j\rangle$ appears in $ready_j$ and j is correct, by I2 at least one correct process signed $\langle ts, v\rangle_i$ and added it to its echo register. Let p_1 be the first correct process to do so, and let t_1 be the moment of adding $\langle ts, v\rangle_i$ to $echo_{p_1}$ (see Figure 2 for illustration). By Observation 8, it is not deleted from the register. Similarly, let p_2 be the first correct process to add $\langle ts, w\rangle_i$ to $echo_{p_2}$ at time t_2. WLOG, $t_1 \geq t_2$. In addition, let p_3 be the first correct process to add $\langle ready, \langle ts, v\rangle_i\rangle$ to $ready_{p_3}$, and let t_3 be the moment of the addition. By I2 it follows that $t_3 > t_1$. By Observation 8, the content of $echo_{p_2}$ and $ready_{p_3}$ is not deleted during the run. By the protocol, at some point in time between t_1 and t_3, p_3 executes line 19 and reads all echo registers. Let $t_1 < t^* < t_3$ be the time when p_3 reads $echo_{p_2}$. Since $t_1 \geq t_2$ we conclude that $t^* > t_2$. Since, p_3 does not see a conflicting value in $echo_{p_2}$, we get that $v = w$.

I4: By I1 at least one correct process j signed $\langle ready, \langle ts, v\rangle_i\rangle$ and added it to $ready_j$ and at least one correct process j' signed $\langle ready, \langle ts, w\rangle_i\rangle$ and added it to $ready_{j'}$. Thus, by I3 $v = w$. ◀

Let us examine an execution E of the algorithm. Let H be the history of E. First, we define H^c to be the history H after removing any pending deliver operations and any pending broadcast operations that did not complete line 11 (which is called from line 6). We define H' to be an augmentation of $H^c|correct$ as follows. For every Byzantine process j and a value v such that v is returned by *deliver$_i$(j,ts)* for at least one correct process i, we add to H' a *broadcast$_j$(ts,v)* operation that begins and ends immediately before the first correct process adds $\langle\langle ts, v\rangle_j, \sigma\rangle$ to its delivery register. Since at least one correct process adds this value at line 11, this moment is well-defined. We construct a linearization E' of H' by defining the following linearization points:

- Let o be a *broadcast$_i$(ts,v)* operation by a correct process i that completed line 11. Note that by the code every completed *broadcast* operation completes line 11 exactly once, and operations that do not complete this line are removed from H'. The operation linearizes when $\langle\langle ts, v\rangle_j, \sigma\rangle$ is added for the first time to delivery register of a correct process, which occurs either when i executes line 11 or when another correct process executes line 22 beforehand. By the code, these lines are between the invocation and the return of the broadcast procedure.

- Let o be a *deliver$_i$(j,ts)* operation by a correct process i that completes line 11 and returns $v \neq \bot$ (note that by the code every completed *deliver* operation that returns $v \neq \bot$ completes line 11 exactly once). If i finds $\langle\langle ts, v\rangle_j, \sigma\rangle$ for some value v in some correct process' deliver register at line 10, then the operation linearizes when i first reads $\langle\langle ts, v\rangle_j, \sigma\rangle$ from a correct process. Otherwise, it linearizes at line 11 when i copies the data to $deliver_i$.
- If o is a completed *deliver$_i$(j,ts)* operation by a correct process i that returns $\bot$ it linearizes at the moment of its invocation.
- Every Byzantine *broadcast$_j$(ts,v)* operation by process j linearizes at the moment we added it.

In H' there are no deliver operations by Byzantine processes. The following lemmas prove that E', the linearization of H', satisfies the sequential specification:

▶ **Lemma 11.** *For a given* deliver(j,ts) *operation that returns $v \neq \bot$, there is at least one preceding broadcast operation in E' of the form* broadcast(ts,v) *invoked by process j.*

Proof. Let o be a *deliver$_i$(j,ts)* operation invoked by a correct process i that returns $v \neq \bot$. Let t be the time when $\langle\langle ts, v\rangle_j, \sigma\rangle$ is added for the first time to a delivery register of a correct process (where σ contains $f+1$ ready signatures). If j is correct then by Lemma 9 j previously invoked *broadcast(ts,v)* and that broadcast linearizes at time t. If j is Byzantine then *broadcast(ts,v)* by process j is added to H' immediately before t. There are two options to the linearization point of o. If i finds $\langle\langle ts, v\rangle_j, \sigma\rangle$ in some correct process' deliver register at line 10, then o linearizes when i first reads $\langle\langle ts, v\rangle_j, \sigma\rangle$ from a correct process and thus it is after time t. Otherwise, it linearizes at line 11 when i copies the data to $deliver_i$, which is also no earlier than time t. ◀

▶ **Lemma 12.** *For a* broadcast$_i$(ts,v) *in E', there does not exist any* broadcast$_i$(ts,w) *in E' for $v \neq w$.*

Proof. If i is a correct process, the proof follows from Assumption 2. If i is Byzantine, *broadcast$_i$(ts,v)* is added immediately before the first correct process adds $\langle\langle ts, v\rangle_i, \sigma\rangle$ to its delivery register. By I4, no correct processes add $\langle\langle ts, w\rangle_i, \sigma\rangle$ to their delivery register for $v \neq w$ and *broadcast$_i$(ts,w)* does not appear in E'. ◀

▶ **Lemma 13.** *For a given* deliver(j,ts) *operation that returns $\bot$, there is no preceding broadcast operation in H' of the form* broadcast(ts,v) *invoked by process j, for $v \neq \bot$.*

Proof. Let o be a *deliver(j,ts)* operation invoked by a correct process i that returns $\bot$. Assume by way of contradiction that there is a preceding *broadcast(ts,v)* operation in H' invoked by process j, for $v \neq \bot$. By definition, the broadcast linearizes no later than the first adding of $\langle\langle ts, v\rangle_j, \sigma\rangle$ to a delivery register of a correct process. Thus, since o linearizes at the moment of its invocation, it sees $\langle\langle ts, v\rangle_j, \sigma\rangle$ at some process' delivery register and returns $v \neq \bot$, in contradiction. ◀

Next, we prove f-resilience.

▶ **Lemma 14.** *(Liveness) Every correct process that invokes some operation eventually returns.*

Proof. If a correct process i invokes a deliver operation then by the code it returns in a constant time. If it invokes *broadcast(ts,v)*, it copies $\langle ts, v\rangle_i$ to $send_i$. By Assumption 1, all correct processes infinitely often call the reliable broadcast API and specifically the refresh procedure, see $\langle ts, v\rangle_i$ and copy it to their echo registers. As signatures are unforgable and i is correct they do not find $\langle ts, w\rangle_i$ for any other $w \neq v$ in any other echo registers and copy a signed $\langle ready, \langle ts, w\rangle_i\rangle$ to their ready registers. By I8, eventually they all see $\langle ready, \langle ts, w\rangle_i\rangle$ in $f+1$ ready registers and copy $\langle ts, w\rangle_i$ to their deliver registers. Eventually $f+1$ correct processes have $\langle ts, w\rangle_i$ in their deliver registers, and since the signatures are valid, the check at line 10 evaluates to true, and i returns v and finish the repeat loop. ◀

We conclude the following theorem:

▶ **Theorem 15.** *Algorithm 1 implements an f-resilient Byzantine linearizable reliable broadcast object for any $f < \frac{n}{2}$.*

7 Byzantine Linearizable Snapshot

In this section, we utilize a reliable broadcast primitive to construct a Byzantine snapshot object with resilience $n > 2f$.

7.1 Snapshot Object

A snapshot [2] is represented as an array of n shared single-writer variables that can be accessed with two operations: *update(v)*, called by process i, updates the i^{th} entry in the array and *snapshot* returns an array. The sequential specification of an atomic snapshot is as follows: the i^{th} entry of the array returned by a *snapshot* invocation contains the value v last updated by an *update(v)* invoked by process i, or its variable's initial value if no update was invoked.

Following Lemma 5, we again must require that correct processes perform operations infinitely often. For simplicity, we require that they invoke infinitely many snapshot operations; if processes invoke either snapshots or updates, we can have each update perform a snapshot and ignore its result.

▶ **Assumption 3.** *All correct processes invoke snapshot operations infinitely often.*

7.2 Snapshot Algorithm

Our pseudo-code is presented in Algorithms 2 and 3. During the algorithm, we compare snapshots using the (partial) coordinate-wise order. That is, let s_1 and s_2 be two n-arrays. We say that $s_2 > s_1$ if $\forall i \in [n]$, $s_2[i].ts > s_1[i].ts$.

Recall that all processes invoke snapshot operations infinitely often. In each snapshot instance, correct processes start by collecting values from all registers and broadcasting their collected arrays in "start" messages (message with timestamp 0). Then, they repeatedly send the identities of processes from which they delivered start messages until there exists a round such that the same set of senders is received from $f+1$ processes in that round. Once this occurs, it means that the $f+1$ processes see the exact same start messages and the snapshot is formed as the supremum of the collects in their start messages.

We achieve optimal resilience by waiting for only $f+1$ processes to send the same set. Although there is not necessarily a correct process in the intersection of two sets of size $f+1$, we leverage the fact that reliable broadcast prevents equivocation to ensure that nevertheless,

Algorithm 2 Byzantine Snapshot: code for process i

```
shared SWMR registers: $\forall j \in [n]$ $collected_i[j] \in \{\bot\} \cup \{\mathbb{N} \times Vals\}$ with selectors $ts$ and
val, initially $\bot$
$\forall k \in \mathbb{N}$, $savesnap_i[k] \in \{\bot\} \cup \{$array of $n$ $Vals \times$ set of messages$\}$ with selectors $snap$
and proof, initially $\bot$
local variables: $ts_i \in \mathbb{N}$, initially 0
$\forall j \in [n]$, $rts_i[j] \in \mathbb{N}$, initially 0
$r, auxnum \in \mathbb{N}$, initially 0
$p \in [n]$, initially 1
$\forall j \in [n], k \in \mathbb{N}$, $seen_i[j][k]$,$senders_i \in \mathcal{P}(\Pi)$, initially $\emptyset$
$\sigma \leftarrow \emptyset$ set of messages

1:  procedure UPDATE($v$)
2:      for $j \in [n]$ do                                  ▷ collect current memory state
3:          update-collect($collected_j$)
4:      $ts_i \leftarrow ts_i + 1$
5:      $collected_i[i] \leftarrow \langle ts_i, v\rangle_i$              ▷ update local component of collected

6:  procedure SNAPSHOT
7:      for $j \in [n]$ do                                  ▷ collect current memory state
8:          update-collect($collected_j$)
9:      $c \leftarrow collected_i$
10:     repeat
11:         $auxnum \leftarrow auxnum + 1$
12:         $snap \leftarrow$ snapshot-aux($auxnum$)
13:     until $snap \geq c$                      ▷ snapshot is newer than the collected state
14:     return $snap$

15: procedure UPDATE-COLLECT(c)
16:     for $k \in [n]$ do
17:         if $c[k].ts > collected_i[k].ts$ and $c[k]$ is signed by $k$ then
18:             $collected_i[k] \leftarrow c[k]$
```

there is a common *message* in the intersection, so two snapshots obtained in the same round are necessarily identical. Moreover, once one process obtains a snapshot s, any snapshot seen in a later round exceeds s.

Each process i collects values from all processes' registers in a shared variable $collect_i$. When starting a snapshot operation, each process runs update-collect, where it updates its collect array (line 8) and saves it in a local variable c (line 9). When it does so, it updates the i^{th} entry to be the highest-timestamped value it observes in the i^{th} entries of all processes' collect arrays (lines $16 - 18$). Then, it initiates the snapshot-aux procedure with a new auxnum tag. Snapshot-aux returns a snapshot, but not necessarily a "fresh" one that reflects all updates that occurred before *snapshot* was invoked. Therefore, snapshot-aux is repeatedly called until it collects a snapshot s such that $s \geq c$, according to the snapshots partial order (lines $10 - 13$).

By Assumption 3 and since the *auxnum* variable at each correct process is increased by 1 every time snapshot-aux is called, all correct processes participate in all instances

Algorithm 3 Byzantine Snapshot auxiliary procedures: code for process i

```
19: procedure MINIMUM-SAVED(auxnum)
20:     $S \leftarrow \{s | \exists j \in [n], s = savesnap_j[auxnum].snap$ and $savesnap_j[auxnum].proof$ is a valid proof of $s\}$
21:     if $S = \emptyset$ then
22:         return $\perp$
23:     $res \leftarrow$ infimum($S$)                ▷ returns the minimum value in each index
24:     $savesnap_i[auxnum] \leftarrow \langle res, \bigcup_{j \in [n]} savesnap_j[auxnum].proof \rangle$
25:     update-collect($res$)
26:     return $res$

27: procedure SNAPSHOT-AUX(auxnum)
28:     initiate new reliable broadcast instance
29:     $\sigma \leftarrow \emptyset$
30:     for $j \in [n]$ do                ▷ collect current memory state
31:         update-collect($collected_j$)
32:     $senders_i \leftarrow \{i\}$                ▷ start message contains collect
33:     broadcast(0,$\langle collect_i \rangle_i$)
34:     while true do
35:         $cached \leftarrow$ minimum-saved($auxnum$)                ▷ check if there is a saved snapshot
36:         if $cached \neq \perp$ then return $cached$
37:         $p \leftarrow (p + 1) \bmod n + 1$                ▷ deliver messages in round robin
38:         $m \leftarrow$ deliver($p, rts_i[p]$)                ▷ deliver next message from $p$
39:         if $m = \perp$ then continue
40:         if $rts_i[p]$= 0 and $m$ contains a signed collect array $c$ then
                                                ▷ start message (round 0)
41:             $\sigma \leftarrow \sigma \cup \{m\}$
42:             update-collect($c$)
43:             $senders_i \leftarrow senders_i \cup \{j\}$
44:         else if $m$ contains a signed set of processes, $jsenders$ then
                                                ▷ round $r$ message for $r > 0$
45:             if $jsenders \nsubseteq senders_i$ then
46:                 continue                ▷ cannot process message, its dependencies are missing
47:             $\sigma \leftarrow \sigma \cup \{m\}$
48:             $seen_i[j][rts_i[p]] \leftarrow jsenders \cup seen_i[j][rts_i[p] - 1]$
49:         $rts_i[p] \leftarrow rts_i[p] + 1$

50:         if received $f + 1$ round-$r$ messages for the first time then
51:             $r \leftarrow r + 1$
52:             broadcast($r$, $\langle senders_i \rangle_i$)

53:         if $\exists s$ s.t. $|\{j|\ seen_i[j][s] = senders_i\}| = f + 1$ then                ▷ stability condition
54:             $r \leftarrow 0$
55:             $senders_i \leftarrow \emptyset$
56:             $\forall j \in [n], k \in \mathbb{N}$, $seen_i[j][k] \leftarrow \emptyset$
57:             $cached \leftarrow$ minimum-saved($auxnum$)                ▷ re-check for saved snapshot
58:             if $cached \neq \perp$ then return $cached$
59:             $savesnap_i[auxnum] \leftarrow \langle collect_i, \sigma \rangle$
                        ▷ $\sigma$ contains all received messages in this snapshot-aux instance
60:             return $collect_i$
```

of snapshot-aux. When a correct process invokes a snapshot-aux procedure with auxnum, it first initiates a new reliable broadcast instance at line 28, dedicated to this instance of snapshot-aux. Note that although processes invoke one snapshot-aux at a time, they may engage in multiple reliable broadcast instances simultaneously. That is, they continue to partake in previous reliable broadcast instances after starting a new one. As another preliminary step of snapshot-aux, each correct process once again updates its collect array using the update-collect procedure (lines 30– 31) and broadcasts it to all processes at line 33. During the execution, a correct processes delivers messages from all other processes in a round robin fashion. The local variable p represents the process from which it currently delivers. In addition, $rts[p]$ maintains the next timestamp to be delivered from p (lines 38, 49, 37). Note that if the delivered message at some point is $\perp$, $rts[p]$ is not increased, so all of p's messages are delivered in order (line 39).

Snapshot-aux proceeds in rounds, which are reflected in the timestamps of the messages broadcast during its execution. Each correct process starts snapshot-aux at round 0, where it broadcasts its collected array; we refer to this as its start message. It then continues to round $r+1$ once it has delivered $f+1$ round r messages (line 51). Each process maintains a local set $senders$ that contains the processes from which it received start messages (line 43). In every round (from 1 onward) processes send the set of processes from which they received start messages (line 52).

Process i maintains a local map $seen[j][r]$ that maps a process j and a round r to the set of processes that j reported to have received start messages from in rounds 1–r (line 48), but only if i has received start messages from all the reported processes (line 45). By doing so, we ensure that if for some correct process i and a round r $seen_i[j][r]$ contains a process l, l is also in $senders_i$. If this condition is not satisfied, the delivered counter for j ($rts[j]$) is not increased and this message will be repeatedly delivered until the condition is satisfied.

Once there is a process i such that there exists a round s and there is a set S of $f+1$ processes j for which $seen_i[j][s]$ is equal to $senders_i$, we say that the *stability condition* at line 53 is satisfied for S. At that time, i and f more processes agree on the collected arrays sent at round 0 by processes in $senders_i$, and $collect_i$ holds the supremum of those collected arrays. This is because whenever it received a start message, it updated its collect so that currently $collect_i$ reflects all collects sent by processes in $senders_i$. Thus, i can return its current collect as the snapshot-aux result. Since reliable broadcast prevents Byzantine processes from equivocating, there are f more processes that broadcast the same $senders$ set at that round, and any future round will "see" this set. As we later show, after at most $n+1$ rounds, the stability condition holds and hence the size of $seen$ is $O(n^3)$. Together with the collected arrays, the total space complexity is cubic in n.

To ensure liveness in case some correct processes complete a snapshot-aux instance before all do, we add a helping mechanism. Whenever a correct process successfully completes snapshot-aux, it stores its result in a savesnap map, with the auxnum as the key (either at line 24 or at line 59). This way, once one correct process returns from snapshot-aux, others can read its result at line 35 and return as well. To prevent Byzantine processes from storing invalid snapshots, each entry in the savesnap map is a tuple of the returned array and a proof of the array's validity. The proof is the set of messages received by the process that stores its array in the current instance of snapshot-aux. Using these messages, correct processes can verify the legitimacy of the stored array. If a correct process reads from savesnap a tuple with an invalid proof, it simply ignores it.

7.3 Correctness

We outline the key correctness arguments highlighting the main lemmas. Formal proofs of auxiliary lemmas appear in Appendix B. To prove our algorithm is Byzantine linearizable, we first show that all returned snapshots are totally ordered (by coordinate-wise order):

▶ **Lemma 16.** *If two snapshot operations invoked by correct processes return s_i and s_j, then $s_j \geq s_i$ or $s_j < s_i$.*

Based on this order, we define a linearization. Then, we show that our linearization preserves real-time order, and it respects the sequential specification. We construct the linearization E as follows: First, we linearize all snapshot operations of correct processes in the order of their return values. Then, we linearize every update operation by a correct process immediately before the first snapshot operation that “sees” it. We say that a snapshot returning s *sees* an update by process j that has timestamp ts if $s[j].ts \geq ts$. If multiple updates are linearized to the same point (before the same snapshot), we order them by their start times. Finally, we add updates by Byzantine processes as follows: We add *update(v)* by a Byzantine process j if there is a linearized snapshot that returns s and $s[j].val = v$. We add the update immediately before any snapshot that sees it.

We next prove that the linearization respects the sequential specification.

▶ **Lemma 17.** *The i^{th} entry of the array returned by a* snapshot *invocation contains the value v last updated by an* update(v) *invoked by process i in E, or its variable's initial value if no update was invoked.*

Proof. Let v be the value in the i^{th} entry of the array returned by a *snapshot*, with a corresponding timestamp ts_v. By the definition of E, *update(v)* by process i with timestamp ts is linearized immediately before $ts_v \geq ts$. If i is correct and multiple update operations by i are linearized at that point, then since i invokes updates sequentially and by Lemmas 26 and 27 their start times are ordered according to the increasing timestamps. Thus, as updates are linearized by their start times, v matches the value of the last update. If i is Byzantine, since we add updates only for values at the moment they are seen, v must match the value of the last update. Additionally, if v is an initial value, then no updates were linearized before it in E. ◀

Because an update is linearized immediately before some snapshot sees it and snapshots are monotonically increasing, all following snapshots see the update as well. Next, we prove in the two following lemmas that E preserves the real-time order.

▶ **Lemma 18.** *If a snapshot operation invoked by a correct process i with return value s_i precedes a snapshot operation invoked by a correct process j with return value s_j, then $s_i \leq s_j$.*

Proof. Assume i invokes snapshot operation $snap_i$, which returns s_i before j invokes snapshot $snap_j$, returning s_j. Let c_1 be the value of $collect_i$ that j reads at line 8 of $snap_j$ and let c_2 be the value it writes in $collect_j$ at line 9. At the end of the last snapshot-aux in $snap_i$, $collected_i \geq s_i$ either because the return value is $collected_i$ (if snapshot-aux returns at line 60), or because s_i is reflected in collect by the end of line 25 if it is a savesnap returned at line 36 or at line 58. Due to the monotonicity of collects (Lemma 27), $s_i \leq c_1$. Because j reads c_1 when calculating c_2, $c_1 \leq c_2$. Finally, by Observation 28, $c_2 \leq s_j$ and by transitivity we get that $s_i \leq s_j$. ◀

▶ **Lemma 19.** *Let s be the return value of a snapshot operation $snap_i$ invoked by a correct process i. Let $update_j(v)$ be an update operation invoked by a correct process j that writes $\langle ts, v\rangle$ and completes before $snap_i$ starts. Then, $s[j].ts \geq ts$.*

Proof. Let t_1 be the time when j completes line 5 in $update_j(v)$ and writes $\langle ts, v\rangle$. Let t_2 be the time when i reads $collect_j[i]$ at line 8 in $snap_i$. By Lemmas 26 and 27, since j is correct, it follows that $collect_j[j].ts \geq ts$ at time $t_2 \geq t_1$. Thus, after line 9 in $snap_i$ $collect_i[j].ts \geq ts$ and by Observation 28, $s[j].ts \geq ts$.

◀

It follows from Lemma 19 and the definition of E, that if an update precedes a snapshot it is linearized before it, and from Lemma 18 that if a snapshot precedes a snapshot it is also linearized before it. The following lemma ensures that if an update precedes another update it is linearized before it. That is, if a snapshot operation sees the second update, it sees the first one.

▶ **Lemma 20.** *If update1 by process i precedes update2 by process j and a snapshot operation snap by a correct process sees update2, then snap sees update1 as well.*

Proof. Let s be the return value of a snapshot that sees update2. By Observation 30, s is the supremum of *collect* arrays sent at line 33. If s sees update2, by Lemma 26, it means that s reflects $collect_j$ after line 5 of update2. After, j performed line 3 and update1 was reflected in $collect_j$. Hence, s sees update1 as well. ◀

Finally, the next lemmas prove the liveness of our algorithm.

▶ **Lemma 21.** *Every correct process that invokes snapshot-aux(auxnum) eventually returns.*

Proof. Assume by induction on auxnum that all snapshot-aux instances with $k' < k$ (if any) have returned at all correct processes. Then, for auxnum=k, all correct processes initiate reliable broadcast instances and broadcast $\langle 0, c\rangle$. This is because all correct processes invoke snapshot infinitely often. Since all messages by correct processes are eventually delivered, they all eventually complete line 50 in each round. Because $|senders|$ is bounded, eventually the $senders$ sets of all correct processes stabilize, and due to reliable broadcast, they contain the same set of processes for all correct processes. Thus, there is a round r for which the condition at line 53 is satisfied. Therefore, at least one correct process returns from snapshot-aux at line 60 (if it did not return sooner). Before returning, it updates its savesnap register at line 59. If it returns at line 36 or at line 58 it also updates its savesnap register at line 24. Every other correct process that has not yet returned from snapshot-aux will read the updated savesnap in the next while iteration and will return at line 36. ◀

▶ **Lemma 22.** *(Liveness) Every correct process that invokes some operation eventually returns.*

Proof. If a correct process i invokes an update operation then by the code it returns in constant time. If i invokes a snapshot operation at time t, let c be the collected array at line 8. Additionally, let k be the maximum *auxnum* of any snapshot-aux operation that was initiated by some process before time t. By Lemma 21, all snapshot-aux invocations eventually return. At snapshot-aux($k+1$), all correct processes see c at lines 30–31 when they update their collect. Since the return value is the supremum of $f+1$ collect arrays, it is guaranteed that when i executes snapshot-aux($k+1$), the returned value res will satisfy $res \geq c$. ◀

We conclude the following theorem:

▶ **Theorem 23.** *Algorithm 2 implements an f-resilient Byzantine linearizable snapshot object for any $f < \frac{n}{2}$.*

Proof. Lemma 16 shows that there is a total order on snapshot operations. Using this order, we have defined a linearization E that satisfies the sequential specification (Lemma 17). We then proved that E also preserves real-time order (Lemmas 18 − 20). Thus, Algorithm 2 is Byzantine linearizable. In addition, Lemma 22 proves that Algorithm 2 is f-resilient. ◀

8 Conclusions

We have studied shared memory constructions in the presence of Byzantine processes. To this end, we have defined Byzantine linearizability, a correctness condition suitable for shared memory algorithms that can tolerate Byzantine behavior. We then used this notion to present both upper and lower bounds on some of the most fundamental components in distributed computing.

We proved that atomic snapshot, reliable broadcast, and asset transfer are all problems that do not have f-resilient emulations from registers when $n \leq 2f$. On the other hand, we have presented an algorithm for Byzantine linearizable reliable broadcast with resilience $n > 2f$. We then used it to implement a Byzantine snapshot with the same resilience. Among other applications, this Byzantine snapshot can be utilized to provide a Byzantine linearizable asset transfer. Thus, we proved a tight bound on the resilience of emulations of asset transfer, snapshot, and reliable broadcast.

Our paper deals with feasibility results and does not focus on complexity measures. In particular, we assume unbounded storage in our constructions. We leave the subject of efficiency as an open question for future work.

References

1 Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

2 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM (JACM)*, 40(4):873–890, 1993.

3 Yehuda Afek, David S Greenberg, Michael Merritt, and Gadi Taubenfeld. Computing with faulty shared objects. *Journal of the ACM (JACM)*, 42(6):1231–1274, 1995.

4 Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of rdma on agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 409–418, 2019.

5 Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra Marathe, and Igor Zablotchi. The impact of rdma on agreement, 2021. `arXiv:1905.12143`.

6 Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Efficient atomic snapshots using lattice agreement. In *International Workshop on Distributed Algorithms*, pages 35–53. Springer, 1992.

7 Alex Auvolat, Davide Frey, Michel Raynal, and François Taïani. Money transfer made simple: a specification, a generic algorithm, and its proof. *Bulletin of EATCS*, 3(132), 2020.

8 Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 2019.

9 Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

10 Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

11 Miguel Castro, Barbara Liskov, et al. A correctness proof for a practical byzantine-fault-tolerant replication algorithm. Technical report, Technical Memo MIT/LCS/TM-590, MIT Laboratory for Computer Science, 1999.

12 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

13 Vicent Cholvi, Antonio Fernandez Anta, Chryssis Georgiou, Nicolas Nicolaou, and Michel Raynal. Atomic appends in asynchronous byzantine distributed ledgers. In *2020 16th European Dependable Computing Conference (EDCC)*, pages 77–84. IEEE, 2020.

14 Shir Cohen and Idit Keidar. Tame the wild with byzantine linearizability: Reliable broadcast, snapshots, and asset transfer. In *35th International Symposium on Distributed Computing*, page 1, 2021.

15 Daniel Collins, Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Matteo Monti, Matej Pavlovic, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, Andrei Tonkikh, and Athanasios Xygkis. Online payments by merely broadcasting messages. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 26–38. IEEE, 2020.

16 Giuseppe Antonio Di Luna, Emmanuelle Anceaume, and Leonardo Querzoni. Byzantine generalized lattice agreement. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 674–683. IEEE, 2020.

17 Rachid Guerraoui, Petr Kuznetsov, Matteo Monti, Matej Pavlovič, and Dragos-Adrian Seredinschi. The consensus number of a cryptocurrency. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 307–316, 2019.

18 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

19 Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM (JACM)*, 45(3):451–500, 1998.

20 Barbara Liskov and Rodrigo Rodrigues. Byzantine clients rendered harmless. In *International Symposium on Distributed Computing*, pages 487–489. Springer, 2005.

21 Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In *International Symposium on Distributed Computing*, pages 311–325. Springer, 2002.

22 Achour Mostéfaoui, Matoula Petrolia, Michel Raynal, and Claude Jard. Atomic read/write memory in signature-free byzantine asynchronous message-passing systems. *Theory of Computing Systems*, 60(4):677–694, 2017.

23 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2009.

24 Rodrigo Rodrigues and Barbara Liskov. Rosebud: A scalable byzantine-fault-tolerant storage architecture. Technical report, 2003.

25 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

26 Xiong Zheng and Vijay K. Garg. Byzantine lattice agreement in asynchronous systems. In Quentin Bramas, Rotem Oshman, and Paolo Romano, editors, *24th International Conference on Principles of Distributed Systems, OPODIS 2020, December 14-16, 2020, Strasbourg, France (Virtual Conference)*, volume 184 of *LIPIcs*, pages 4:1–4:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.OPODIS.2020.4`.

Appendix A Byzantine Asset Transfer

In this section we adapt the asset transfer implementation from snapshots given in [17] to a Byzantine asset transfer. The algorithm is very simple. It is based on a shared snapshot array S, with a cell for each client process i, representing i's outgoing transactions. An additional immutable array holds all processes' initial balances. A process i's balance is computed by taking a snapshot of S and applying all of i's valid incoming and outgoing transfers to i's initial balance. A transfer invoked by process i checks if i's balance is sufficient, and if so, appends the transfer details (source, destination, and amount) to i's cell. Similarly to the use of dependencies in the (message-passing broadcast-based) asset transfer algorithm of [17], we also track the history of every transaction. To this end, we append to the process's cell also the snapshot taken to compute the balance for each transaction.

▶ **Theorem 24.** *Algorithm 4 implements an f-resilient Byzantine linearizable asset transfer object for any $f < \frac{n}{2}$.*

Proof. At any point during a sequential execution, we denote by $B(p)$ the balance of process p. Recall that the operation $transfer(src, dst, amount)$ causes the following changes: $B(src) = B(src) - amount$ and $B(dst) = B(dst) + amount$.

In addition, at any point during a concurrent execution, we represent by $balance(p)$ the balance of process p derived from the state as follows:

If p is a correct process:

$$
\begin{aligned}
balance(p) \overset{def}{=} initial(p) & \\
& + \sum_{j \in correct(\Pi)} amount \mid txn = \langle *, j, p, amount, * \rangle \in S[j] \wedge valid(txn) \\
& + \sum_{j \in Byzantine(\Pi)} amount \mid txn = \langle *, j, p, amount, * \rangle \in S[j] \wedge valid(txn) \\
& \wedge \text{txn was read by some correct process} \\
& - \sum_{j \in \Pi} amount \mid txn = \langle *, p, j, amount, * \rangle \in S[p] \wedge valid(txn)
\end{aligned}
$$

If p is a Byzantine process:

$$
\begin{aligned}
balance(p) \overset{def}{=} initial(p) & \\
& + \sum_{j \in correct(\Pi)} amount \mid txn = \langle *, j, p, amount, * \rangle \in S[j] \wedge valid(txn) \\
& + \sum_{j \in Byzantine(\Pi)} amount \mid txn = \langle *, j, p, amount, * \rangle \in S[j] \wedge valid(txn) \\
& \wedge \text{txn was read by some correct process} \\
& - \sum_{j \in \Pi} amount \mid txn = \langle *, p, j, amount, * \rangle \in S[p] \wedge valid(txn) \\
& \wedge \text{txn was read by some correct process}
\end{aligned}
$$

Let us examine an execution E of the algorithm. Let H be the history of E. First, we define H^c to be the history H after removing any pending read operations and any pending transfer operations that did not complete line 18. We define H' to be an augmentation of $H^c|correct$ as follows.

Algorithm 4 Byzantine Asset Transfer: code for process i

```
shared Byzantine snapshot: $S$
initial– immutable array of initial balances
local variables: $txns_i$ – sets of outgoing transaction, initially {}
$ts_i \in \mathbb{N}$, initially 0
$snap$ – array of sets of transactions, initially array of empty sets    ▷ the last snapshot taken

struct $txn$ contains:
    timestamp ts,
    source src,
    destination dst,
    amount amount

procedure BALANCE(j,snap)
    $incoming \leftarrow 0$
    $outgoing \leftarrow 0$
    for $l \in [n]$ do
        for $k \in snap[l]$ do
            if $snap[l][k].dst = j$ and valid$(snap[l][k])$ then
                $incoming \leftarrow incoming + snap[l][k].amount$
    for $k \in snap[j]$ do
        if valid$(snap[j][k])$ then
            $outgoing \leftarrow outgoing + snap[j][k].amount$
    return $initial(j) + incoming - outgoing$

procedure TRANSFER(src,dst,amount)
    $ts_i \leftarrow ts_i + 1$
    $snap \leftarrow S.snapshot()$
    if $balance(src, snap) < amount$ then
        return false
    $txns_i \leftarrow txns_i.append(\langle ts_i, src, dst, amount, snap\rangle_i)$
    $S.update(txns_i)$
    return true

procedure READ(j)
    $snap \leftarrow S.snapshot()$
    return $balance(j, snap)$
```

For every Byzantine process j and a transaction $txn = (ts, j, dst, amount, deps)$ such that txn appears in the array returned by the snapshot procedure (either in line 21 or line 14) for at least one correct process i, we add to H' a *transfer*$_j$*(j,dst,amount)* operation that begins and ends immediately before the first correct process performs that snapshot procedure. Since at least one correct process reads this transaction, this moment is well-defined. We construct a linearization E' of H' by defining the following linearization points:

- Let o be a *read*$_i$*(j)* operation by a correct process i that completes line 14. The operation linearizes at that moment.
- Let o be a *transfer*$_i$*(i,dst,amount)* operation by a correct process i that completed line 18. The operation linearizes at that moment. Note that operations that do not complete this line are removed from H'. By the code, these lines are between the invocation and the return of the broadcast procedure.
- If o is a completed *transfer*$_i$*(i,dst,amount)* operation by a correct process i that returns false it linearizes at line 21.
- Every Byzantine *transfer*$_j$*(j,dst,amount)* operation by process j linearizes at the moment we added it.

In H' there are no read operations by Byzantine processes. It is clear from construction that each operation invoked by a correct process is mapped to some point between its invocation event and its response event. We now prove that the concrete concurrent run simulates the specification. That is, if we execute the sequential run defined by the linearization points the changes in the balances (represented by B) reflects the actual changes on *balance*. Before the execution begins, $B(p)$ is the initial balance of process p. As the snapshot is empty before the run begins, it holds by definition that $B(p) = balance(p)$. We now show that at any point $B(p) = balance(p)$.

We prove the equivalence of $B(p)$ and $balance(p)$ by induction on the steps in the executions. We assume that the claim holds before a particular step and show that it remains the same after each step. For a correct process p, $balances(p)$ changes at line 18 when some transfer involving p is updated in the snapshot. As this is the linearization point of a transfer operation, the same change in balance also applies to $B(p)$ at that moment. For a Byzantine process p, $balances(p)$ changes at line 14 or line 14 when its transaction is being read by a correct process. A transfer operation by Byzantine processes is added immediately before the first correct process reads it, so this change also reflect $B(p)$ at that moment.

Next, we prove f-resilience.

▶ **Lemma 25.** *(Liveness) Every correct process that invokes some operation eventually returns.*

Proof. This is immediate from the snapshot f-resilient guarantees and the fact that all other operations are local computations. ◀

◀

Appendix B Byzantine Snapshot: Correctness

▶ **Lemma 26.** *For a correct process i, at each point during an execution $collect_i[i]$ contains the value signed by j with the highest timestamp until that point.*

Proof. By induction on the execution; $collect_i[i]$ can change either at line 5 or at line 18. If it changes at line 5, ts_i is increased and $collect_i[i]$ contains the value with the highest timestamp. By induction, no signed value encountered at line 17 has s timestamp higher than the one in $collect_i[i]$, so it is not updated at line 18. ◀

▶ **Lemma 27.** *For a correct process i, $collect_i$ is monotonically increasing.*

Proof. Let $j \in [n]$. We prove that every time the value in $collect_i[j]$ is updated from m to m', it holds that $m'.ts > m.ts$. By the code $collect_i[j]$ changes either at line 5 or at line 18. In both cases, the value in $collect_i[j]$ is signed by j. If $collect_i[j]$ changes at line 18, then monotonicity is immediate from the condition at line 17. Otherwise, it changes at line 5, indicating that $i = j$ and monotonicity follows from Lemma 26. ◀

▶ **Observation 28.** *For a snapshot operation invoked by a correct process i, let c_i be the collected array at line 8 and let s be the return value. Then, $s \geq c_i$.*

Proof. Immediate from the condition at line 13. ◀

▶ **Invariant 1.** *For any correct process i that invokes snapshot-aux(k), it holds that $collect_i$ is the supremum of the arrays in start message sent by processes in $senders_i$ from line 33 and until the return value of snapshot-aux(k) is determined at line 23 or at line 60.*

Proof. First, at line 33 $senders_i$ contains i itself, and i sends exactly its $collect_i$ array. The argument continues by induction on steps of snapshot-aux(k). Other than line 25, $collect_i$ and $senders_i$ change together: Whenever i receives a start message with an array c from process j, it updates $collect_i$ with the higher-timestamped values found in c and adds j to $senders_i$ (lines 42– 43).

◀

▶ **Definition 29.** *We say that the stability condition holds for a return value s_1 of snapshot-aux(k) with a round r and a set of processes S if (1) $|S| \geq f+1$, (2) there is a set $S' \supseteq S$ so that for each $p \in S$ the union of all jsenders sets sent in p's messages in rounds 1 to r is S', and (3) s_1 is the supremum of the collects sent in start messages of members of S'.*

▶ **Observation 30.** *If s_1 is returned from snapshot-aux(k) by a correct process i, then s_1 satisfies the stability condition for some set S in some round r.*

Proof. Consider two cases. First, if i returns s_1 at line 60, then the condition is satisfied for s_1 with the round s that satisfies the condition at line 53 and the set of $f+1$ processes for which the condition at line 53 holds. S' is the set in $senders_i$ at the time the condition is satisfied. Since messages are delivered in order, we get that $S' \supseteq S$. Because the return value is $collect_i$, (3) follows from Invariant 1.

Second, if i adopts a saved snapshot s_1 with a proof and returns at line 36 or at line 58, then the proof contains $f+1$ messages from some round r and corresponding start messages satisfying the stability condition. ◀

▶ **Lemma 31.** *For a given k, Let i, j be two correct processes that return s_i, s_j from snapshot-aux(k). Then $s_i \leq s_j$ or $s_i > s_j$.*

Proof. By Observation 30, s_i satisfies the stability condition for some set S_1 in some round r_1. Let S'_1 be the set guaranteed from the definition. Also by Observation 30, s_j satisfies the stability condition and some set S_2 in some round r_2. Let S'_2 be the set guaranteed from the definition.

Since $|S_1| \geq f+1$ and $|S_2| \geq f+1$, there is at least one process $p \in S_1 \cap S_2$. Due to reliable broadcast, p cannot equivocate with the set of processes $jsenders$ sent in each round of snapshot-aux(k).

- If $r_1 = r_2$: By property (2) of Definition 29 $S'_1 = S'_2$, and by (3) $s_i = s_j$.
- If $r_1 \neq r_2$: Assume WLOG $r_1 < r_2$. Since the union of all jsenders sets sent in p's messages in rounds 1 to r_2 is a superset of those sent in rounds 1 to r_1, $S_2 \supseteq S_1$ and then by (3) $s_j \geq s_i$.

◀

▶ **Lemma 32.** *Let i, j be two correct processes returning s_i, s_j resp. from snapshot-aux with auxnum $= k$, such that $s_j > s_i$. Then when i begins any snapshot-aux$_i(k')$ for $k' > k$, collect$_i > s_j$.*

Proof. Since j is correct, by Observation 30, s_j satisfies the stability condition. Let t_1 be a time when the condition is satisfied. At time t_1, there is at least one correct process l such that $collect_l \geq s_j$. We show that either (1) j does not return s_j or (2) i begins snapshot-aux$_i(k')$ with $collect_i > s_j$. If i begins snapshot-aux$_i(k')$ after t_1, then when it updates its collect at lines 30–31, it reads the values in $collect_l$. By Lemma 27, $collect_l$ is greater than or equal to its value at time t_1. Thus, we get that $collect_i \geq collect_l \geq s_j$ and (2) holds. Otherwise, i saves s_i at line 59 before starting snapshot-aux(k'), which is before time t_1. Between time t_1 and the time it returns s_j, j checks stored snapshots (at line 21). When it does so, j reads s_i, and since $s_j > s_i$ and j returns the minimal array it sees, (1) holds.

◀

▶ **Lemma 33.** *If snapshot-aux$_i(k)$ of a correct process i returns s_i, there is a correct process j s.t. j invoked snapshot-aux$_j(k)$ and $s_i \geq c_j$, where c_j is the value of collected$_j$ after the collection at line 31 in snapshot-aux(k) at j.*

Proof. If snapshot-aux$_i(k)$ returns at line 60, then i returns $collect_i$ and by Lemma 27, $s_i = collect_i$ is greater than or equal to its value after the collection at line 31 so the lemma holds with $i = j$. Otherwise, snapshot-aux$_i(k)$ returns s_i at line 36 or at line 58 and s_i is an array saved in savesnap with a proof σ signed by process p. Since i validates s_i, there was a round r such that $|\{j| \, seen_p[j][s] = senders_p\}| \geq f+1$. Thus, there was at least one correct process j in this set. Since j adds itself to $senders_j$ (Section 7.1), $senders_j$ is broadcast by j at every round (Section 7.1), and it is the set added to $seen$, the array c_j sent in j's start message is reflected in s_i. This set is exactly the value of $collected_j$ after the collection at line 31 in snapshot-aux(k) at j, and hence $s_i \geq c_j$. ◀

Lemma 16. *If two snapshot operations invoked by correct processes return s_i and s_j, then $s_j \geq s_i$ or $s_j < s_i$.*

Proof. By the code, s_i is the return value of some snapshot-aux$_i(k_i)$ and s_j is the return value of some snapshot-aux$_j(k_j)$. WLOG, $k_i \geq k_j$.

- If $k_i = k_j$, the proof follows from Lemma 31.
- If $k_i > k_j$: By Lemma 33, there is a correct process l that invoked snapshot-aux$_l(k_i)$, collected c_l at line 31 of snapshot-aux$_l(k_i)$ (where c_l is the value of $collected_l$ at that time), and $s_i \geq c_l$. Let s_l be the return value of snapshot-aux$_l(k_j)$ (note that l invokes snapshot-aux with increasing auxnums, so such a value exists). Consider two cases. First,

if $s_j > s_l$, then by Lemma 32, $s_j \leq c_l$. Thus, $s_j \leq c_l \leq s_i$ and the lemma follows. Otherwise, $s_j \leq s_l$. At the end of snapshot-aux$_l(k_j)$ *collected*$_l \geq s_l$ because either the return value is *collected*$_l$, or s_l is reflected in collect by the end of line 25. Due to the monotonicity of collects (Lemma 27), $s_l \leq c_l$. We conclude that $s_j \leq s_l \leq c_l \leq s_i$, as required.

◀

2.3 Be Aware of Your Leaders

Appears in the 26th International Conference of Financial Cryptography and Data Security (FC 2022).

Be Aware of Your Leaders

Shir Cohen
Novi Research, Technion

Rati Gelashvili
Novi Research

Lefteris Kokoris Kogias
Novi Research, IST Austria

Zekun Li
Novi Research

Dahlia Malkhi
Novi Research

Alberto Sonnino
Novi Research

Alexander Spiegelman
Novi Research

Abstract

Advances in blockchains have influenced the State-Machine-Replication (SMR) world and many state-of-the-art blockchain-SMR solutions are based on two pillars: *Chaining* and *Leader-rotation*. A predetermined round-robin mechanism used for Leader-rotation, however, has an undesirable behavior: crashed parties become designated leaders infinitely often, slowing down overall system performance. In this paper, we provide a new Leader-Aware SMR framework that, among other desirable properties, formalizes a *Leader-utilization* requirement that bounds the number of rounds whose leaders are faulty in crash-only executions.

We introduce Carousel, a novel, reputation-based Leader-rotation solution to achieve Leader-Aware SMR. The challenge in adaptive Leader-rotation is that it cannot rely on consensus to determine a leader, since consensus itself needs a leader. Carousel uses the available on-chain information to determine a leader locally and achieves Liveness despite this difficulty. A HotStuff implementation fitted with Carousel demonstrates drastic performance improvements: it increases throughput over 2x in faultless settings and provided a 20x throughput increase and 5x latency reduction in the presence of faults.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases SMR; Leader-election, Chain-quality

1 Introduction

Recently, Byzantine agreement protocols in the eventually synchronous model such as Tendermint [5], Casper FFG [6], and HotStuff [22], brought two important concepts from the world of blockchains to the traditional State Machine Replication (SMR) [12] settings, *Leader-rotation* and *Chaining*. More specifically, these algorithms operate by designating one party as *leader* of each round to propose the next block of transactions that extends a *chained* sequence of blocks. Both properties depart from the approach used by classical protocols such as PBFT [7], Multi-Paxos [13] and Raft [17] (the latter two in benign settings). In those solutions, a stable leader operates until it fails and then it is replaced by a new leader. Agreement is formed on an immutable sequence of indexed (rather than chained) transactions, organized in slots.

Leader-rotation is important in a Byzantine setting, since parties should not trust each other for load sharing, reward management, resisting censoring of submitted transactions, or ordering requests fairly [11]. The advantage of Chaining is that it simplifies the leader handover since in the common case the chain eliminates the need for new leaders to catch up with outcomes from previous slots.

In the permissioned SMR settings [1], most existing Leader-rotation mechanisms use a round-robin approach to rotate leaders [8, 21, 22]. This guarantees that honest parties get a chance to be leaders infinitely often, which is sufficient to drive progress and satisfy *Chain-quality* [10]. Roughly speaking, the latter stipulates that the number of blocks committed to the chain by honest parties is proportional to the honest nodes' percentage. The drawback of such a mechanism is that it does not bound the number of faulty parties which are designated as leaders during an execution. This has a negative effect on latency even in crash-only executions, as each crashed leader delays progress. Similarly to XFT [14], we seek to improve the performance in such executions. Unlike XFT, we also maintain Chain-quality to thwart Byzantine attacks.

In this paper, we propose a leader-rotation mechanism, Carousel, that enjoys both worlds. Carousel satisfies non-zero Chain-quality, and at the same time, bounds the number of faulty leaders in crash-only executions after the global stabilization time (GST), a property we call *Leader-utilization*. The Carousel algorithm leverages Chaining to execute purely locally using information available on the chain, avoiding any extra communication. To capture all requirements, we formalize a *Leader-Aware SMR* problem model, which alongside Agreement, Liveness and Chain-quality, also requires Leader-utilization. We prove that Carousel satisfies the Leader-Aware SMR requirements.

The high-level idea to satisfy Leader-utilization is to track active parties via the records of their participation (e.g. signatures) at the committed chain prefix and elect leaders among them. However, if done naively, the adversary can exploit this mechanism to violate Liveness or Chain-quality. The challenge is that there is no consensus on a committed prefix to determine a leader, since consensus itself needs a leader. Diverging local views on committed prefixes may be effectuated, for instance, by having a Byzantine leader reveal an updated head of the chain to a subset of the honest parties. Hence, Carousel may not have agreement on the leaders of some rounds, but nevertheless guarantees Liveness and Leader-utilization after GST.

To focus on our leader-rotation mechanism, we abstract away all other SMR components by defining an SMR framework. Similarly to [20], we capture the logic and properties of forming and certifying blocks of transactions in each round in a *Leader-based round (LBR)* abstraction, and rely on a Pacemaker abstraction [4, 15, 16] for round synchronization. We prove that when instantiated into this framework, Carousel yields a Leader-Aware SMR protocol. Specifically, we show (1) for Leader-utilization: at most $O(f^2)$ faulty leaders may be elected in crash-only executions (after GST); and (2) for Chain-quality: one out of $O(f)$ blocks is authored by an honest party in the worst-case. Note that in practice Chain-quality guarantees are much better since the worst case scenario requires the adversary to posses an unrealistic power.

We provide an implementation of Carousel in a HotStuff-based system and an evaluation that demonstrates a significant performance improvement. Specifically, we get over 2x throughput increase in faultless settings, and 20x throughput increase and 5x latency reduction in the presence of faults. Our mechanism is adopted in the most recent version of DiemBFT [21], a deployed HotStuff-based system.

2 Model and Problem Definition

We consider a message-passing model with a set of n parties $\Pi = \{p_1, \ldots, p_n\}$, out of which $f < \frac{n}{3}$ are subject to failures. A party is *crashed* if it halts prematurely at some point during an execution. If it deviates from the protocol it is *Byzantine*. An *honest* party never crashes or becomes Byzantine. We say that an execution is *crash-only* if there are no Byzantine failures therein.

For the theoretical analysis we assume an eventually synchronous communication model [9] in which there is a global stabilization time (GST) after which the network becomes synchronous. That is, before GST the network is completely asynchronous, while after GST messages arrive within a known bounded time, denoted as δ.

As we later describe, we abstract away much of the SMR implementation details by defining and using primitives. Therefore, our Leader-rotation solution is model agnostic and the adversarial model depends on the implementation choices for those primitives.

Leader-Aware SMR

In this section we introduce some notation and then define the Leader-Aware SMR problem. Roughly speaking, Leader-Aware SMR captures the desired properties of the Leader-rotation mechanism in SMR protocols that are leader-based.

An SMR protocol consists of a set of parties aiming to maintain a growing chain of *blocks*. Parties participate in a sequence of rounds, attempting to form a block per round. In Leader-Aware SMR, each round is driven by a leader. We capture these rounds via the Leader-based round (LBR) abstraction defined later.

A block consists of transactions and the following meta-data:

- A (cryptographic) link to a *parent* block. Thus, each block implicitly defines a chain to the genesis block.
- A round number in which the block was formed.
- The author id of the party that created the block.
- A certificate that (cryptographically) proves that $2f + 1$ parties endorsed the block in the given round and with the given author. We assume that it is possible to obtain the set of $2f + 1$ endorsing parties[1].

Note that having a round number and the author id as a part of the block is not strictly necessary, but they facilitate formalization of properties and analysis. For example, an *honest block* is defined as a block authored by an honest party and a *Byzantine block* is a block authored by a Byzantine party.

We assume a predicate $\texttt{certified}(B, r) \in \{true, false\}$ that locally checks whether the block has a valid certificate, i.e. it has $2f+1$ endorsements for round r. If $\texttt{certified}(B, r) = true$ we say that B is a *certified* block of round r. When clear from context, we say that B is *certified* without explicitly mentioning the round number.

An SMR protocol does not terminate, but rather continues to form blocks. Each block B determines its *implied* chain starting from B to the genesis block via the parent links. We use notation $B \longrightarrow B'$, saying B' *extends* B, if block B is on B''s implied chain. Honest parties can *commit* blocks in some rounds (but usually not all). A committed block indirectly commits its implied chain. An SMR protocol must satisfy the following:

[1] This can be achieved by multi-signature schemes which are practically as efficient as threshold signatures [3].

▶ **Definition 1** (Leader-Aware SMR).
- ***Liveness:*** *An unbounded number of blocks are committed by honest parties.*
- ***Agreement:*** *If an honest party p_i has committed a block B, then for any block B' committed by any honest party p_j either $B \longrightarrow B'$ or $B' \longrightarrow B$.*
- ***Chain-quality:*** *For any block B committed by an honest party p_i, the proportion of Byzantine blocks on B's implied chain is bounded.*
- ***Leader-Utilization:*** *In crash-only executions, after GST, the number of rounds r for which no honest party commits a block formed in r is bounded.*

The first two properties are common to SMR protocols. While most SMR algorithms satisfy the above mentioned Liveness condition, a stronger Liveness property can be defined, requiring that each honest party commits an unbounded number of blocks. This property can be easily be achieved by an orthogonal forwarding mechanism, where each honest leader that creates a block explicitly sends it to all other parties. A notion of Chain-quality that bounds the adversarial control over chain contents was first suggested by Garay et al. [10]. We introduce the Leader-utilization property to capture the quality of the Leader-rotation mechanism in crash-only executions. Note that although it is tempting to define leader utilization for Byzantine executions as well, it seems impossible to do so without failure detectors. Byzantine parties can decide not to form a block whenever they become leaders. This reduces to the question – can we bound the number of adversarial leaders? the answer is, unfortunately, no.

3 Leader-Aware SMR: The Framework

In order to isolate the Leader-rotation problem in Leader-Aware SMR protocols, we abstract away the remaining logic into two components. First, similar to [19, 20] we capture the logic to form and commit blocks by the *Leader-based round (LBR)* abstraction (Section 3.1). We follow [4, 16] and capture round synchronization by the Pacemaker abstraction (Section 3.2). These two abstractions can be instantiated with known implementations from existing SMR protocols.

In Section 3.3 we define the core API for Leader-rotation and combine it with the above components to construct an SMR protocol. In Section 4 we present a Leader-rotation algorithm that can be easily computed based on locally available information and makes the construction a Leader-Aware SMR.

3.1 Leader-based round (LBR)

The LBR abstraction exposes to each party p_i an API to invoke $LBR(r, \ell)$, where $r \in \mathbb{N}$ is a round number and ℓ is the leader of round r according to party p_i. Intuitively, a leader-based round captures an attempt by parties to certify and commit a block formed by the leader[2] – which naturally requires sufficiently many parties to agree on the identity of the leader. We assume that non-Byzantine parties can only endorse a block B with round number r and author ℓ by calling $LBR(r, \ell)$.

Every LBR invocation returns within $\Delta_l > c\delta$ time, where c depends on the specific LBR implementation (i.e., each round requires a causal chain of c messages to complete). That is, Δ_l captures the inherent timeouts required for eventually synchronous protocols. We say

[2] Existing SMR protocols may have separate rounds (and even leaders) for forming and committing blocks, but this distinction is not relevant for the purposes of the paper and LBR abstraction is defined accordingly.

that round r has $k \leq n$ *LBR-synchronized(ℓ)* invocations if k honest parties invoke $LBR(r, \ell)$ after GST and within $\Delta_l - c\delta$ time of each other with the same party ℓ[3].

The return value of an LBR invocation in round r is always a block with a round number $r' \leq r$. The intention is for LBR invocations to return gradually growing committed chains. Occasionally, there is no progress, in which case the invocations are allowed to return a committed block whose round r' is smaller than r. Formally, the output from LBR satisfies the following properties:

▶ **Definition 2** (LBR).
- ***Endorsement:*** *For any block B and round r, if* `certified`$(B, r) =$ *true, then the set of endorsing parties of B contains $2f + 1$ parties.* [4]
- ***Agreement:*** *If B and B' are certified blocks that are each returned to an honest party from an LBR invocation, then either $B \longrightarrow B'$ or $B' \longrightarrow B$.*
- ***Progress:*** *If there are $k \geq 2f + 1$ LBR-synchronized(ℓ) invocations at round r and ℓ is honest, then they all return a certified B with round number r authored by ℓ.*
- ***Blocking:*** *If a non-Byzantine party ℓ never invokes $LBR(r, \ell)$, then no $LBR(r, \ell)$ invocation may return a certified block formed in round r.*
- ***Reputation:*** *If a non-Byzantine party p never invokes LBR for round r, then any certified block B with round number r does not contain p among its endorsers.*

The LBR definition intends to capture just the key properties required for round abstraction in SMR protocols but leaves room for various interesting behavior. For example, if the progress preconditions are not met at round r, then some honest parties may return a block B for round r while others do not. Moreover, in this case the adversary can *hide* certified blocks from honest parties and reveal them at any point via the LBR return values.

3.2 The Pacemaker

The Pacemaker [4, 15, 16] component is a commonly used abstraction, which ensures that, after GST, parties are synchronized and participate in the same round long enough to satisfy the LBR progress. We assume the following:

▶ **Definition 3** (Pacemaker). *The Pacemaker eventually produces* `new_round(r)` *notifications at honest parties for each round r. Suppose for some round r all* `new_round(r)` *notifications at non-Byzantine parties occur after GST, the first of which occurs at time T_f, and the last of which occurs at time T_l. Then no non-Byzantine party receives a* `new_round(r + 1)` *notification before $T_l + \Delta_p$ and $T_l - T_f \leq \delta$. The Pacemaker can be instantiated with any parameter $\Delta_p > 0$.*

To combine the LBR and Pacemaker components into an SMR protocol in Section 3.3 we fix $\Delta_p = \Delta_l$. Note that by using the above definition, the resulting protocol is not responsive since parties wait Δ_p before advancing rounds. This can easily be fixed by using a more general Pacemaker definitions from [4, 15, 16]. However, we chose the simplified version above for readability purposes since the Pacemaker is orthogonal to the thesis of our paper.

[3] LBR-synchronized requires that the corresponding execution intervals have a shared intersection lasting $\geq c\delta$ time.

[4] Note that Endorsement implies that although LBR can be invoked for round r with more than one leader l, there is at most one author for a block in r.

3.3 Leader-rotation - the missing component

In Algorithm 1 we show how to combine the LBR and Pacemaker abstractions into a leader-based SMR protocol. The missing component is the Leader-rotation mechanism, which exposes a `choose_leader`(r, B) API. It takes a round number $r \in \mathbb{N}$ and a block B and returns a party $p \in \Pi$. The `choose_leader` procedure is locally computed by each honest party at the beginning of every round.

The Agreement property of Algorithm 1 follows immediately from the Agreement property of LBR, regardless of `choose_leader` implementation. In Appendix A we prove that Algorithm 1 satisfies liveness as long as all honest parties follow the same `choose_leader` procedure and that this procedure returns the same honest party at all of them infinitely often. In the next section we instantiate Algorithm 1 with Carousel: a specific `choose_leader` implementation to obtain a Leader-Aware SMR protocol. That is, we prove that Algorithm 1 with Carousel satisfies liveness, Chain-quality, and Leader-utilization.

Algorithm 1 Constructing SMR: code for party p_i

```
1: $commit\_head \leftarrow genesis$
2: upon new_round (r) do
3:     $leader \leftarrow$ choose_leader (r, $commit\_head$)
4:     $B \leftarrow LBR(r,leader)$
5:     if $commit\_head \longrightarrow B$ then
6:         commit $B$          ▷ all blocks in $B$'s implied chain that were not yet committed.
7:         $commit\_head \leftarrow B$
```

4 Carousel: A Novel Leader-Rotation Algorithm

In this section, we present Carousel– our Leader-rotation mechanism. The pseudo-code is given in Algorithm 2, which combined with Algorithm 1 allows to obtain the first Leader-Aware SMR protocol.

We use reputation to avoid crashed leaders in crash-only executions. Specifically, at the beginning of round r, an honest party checks if it has committed a block B with round number $r-1$. In this case, the endorsers of B are guaranteed to not have crashed by round $r-1$. For Chain-quality purposes, the f latest authors of committed blocks are excluded from the set of endorsers, and a leader is chosen deterministically from the remaining set.

If an honest party has not committed a block with round number $r-1$, it uses a round-robin fallback scheme to elect the round r leader. Notice that different parties may or may not have committed a block with round number $r-1$ before round r. In fact, the adversary has multiple ways to cause such divergence, e.g. Byzantine behavior, crashes, or message delays. As a result, parties can disagree on the leader's identity, and potentially compromise liveness. We prove, however, that Carousel satisfies liveness, as well as leader utilization and Chain-quality. Specifically, we show that (1) the number of rounds r for which no honest party commits a block formed in r is bounded by $O(f^2)$; and (2) at least one honest block is committed every $5f+2$ rounds. The argument is non-trivial since, for example, we need to show that the adversary cannot selectively alternate the fallback and reputation schemes to control the Chain-quality.

Algorithm 2 Leader-rotation: code for party p_i

```
8: procedure choose_leader($r$, $commit\_head$)
9:     $last\_authors \leftarrow \emptyset$
10:    if $commit\_head.round\_number \neq r - 1$ then
11:        return ($r \mod n$)                              ▷ round-robin fallback
12:    $active \leftarrow commit\_head.endorsers$
13:    $block \leftarrow commit\_head$
14:    while $|last\_authors| < f \wedge block \neq genesis$ do
15:        $last\_authors \leftarrow last\_authors \cup \{block.author\}$
16:        $block \leftarrow block.parent$
17:    $leader\_candidates \leftarrow active \setminus last\_authors$
18:    return $leader\_candidates.pick\_one()$        ▷ deterministically pick from the set
```

4.1 Correctness

Leader-Utilization.

In this section, we are concerned with the protocol efficiency against crash failures. We consider time after GST, and at most f parties that may crash during the execution but follow the protocol until they crash (i.e., non-Byzantine). We say that a party p crashes in round r if $r+1$ is the minimal number for which p does not invoke LBR in line 4. Accordingly, we say that a party is *alive* at all rounds before it crashes. In addition, we say that a round r occurs after GST if all `new_round` (r) notifications at honest parties occur after GST.

We start by introducing an auxiliary lemma which extends the LBR Progress property for crash-only executions. Since in a crash-only case faulty parties follow the protocol before they crash, honest parties cannot distinguish between an honest leader and an alive leader that has not crashed yet. Hence, the LBR Progress property hold even if the leader crashes later in the execution. Formal proof of the following technical lemma, using indistinguishability arguments, appears in Appendix A.

▶ **Lemma 4.** *In a crash-only execution, let r be a round with $k \geq 2f+1$ LBR-synchronized(ℓ) invocations, such that ℓ is alive at round r, then these k invocations return a certified B with round number r authored by ℓ.*

Furthermore, if no party crashes in a given round and the preconditions of the adapted LBR Progress conditions are met a block is committed in that round and another alive leader is chosen.

▶ **Lemma 5.** *If the preconditions of Lemma 4 hold and no party crashes in round r, then $k \geq 2f+1$ honest parties commit a block for round r and return the same leader ℓ' at line 3 of round $r+1$ and ℓ' is alive at round r.*

Proof. By Lemma 4, k honest parties return from $LBR(r, \ell)$ with a certified block B with round number r authored by ℓ. Then, since $commit\_head \longrightarrow B$, they all commit B at line 6 of round $r+1$. By the LBR Reputation property, the set of B's endorsers does not include parties that crashed in rounds $< r$. Since no party crashes in round r, B's endorsers are all alive in round r. Since these $2f+1$ parties each committed block B with round number r, in `choose_leader` in Algorithm 1, they all use the reputation scheme (line 18) to choose the leader of round $r+1$, that we showed is alive at round r. ◀

Next, we utilize the latter to prove that in a round with no crashes, it is impossible for a minority of honest parties to return with a certified block from an LBR instance. Namely, either no honest party returns a block, or at least $2f + 1$ of them do.

▶ **Lemma 6.** *In a crash-only execution, let r be a round after GST in which no party crashes. If one honest party returns from LBR with a certified block B with round number r, then $2f + 1$ honest parties return with B.*

Proof. Assume an honest party returns a certified block B with round number r after invoking $LBR(r, \ell)$. By the LBR Blocking property, ℓ itself must have invoked $LBR(r, \ell)$ and by assumption it was *alive* at round r. By the LBR Endorsement property, the set of endorsing parties of B contains $2f + 1$ parties. Since we consider a crash-only execution, it follows by assumption that $2f + 1$ party called $LBR(r, \ell)$. Due to the use of Pacemaker, these calls are LBR-synchronized(ℓ) invocations. Finally, by Lemma 4 all these calls return a certified B with round number r authored by ℓ. ◀

We prove that in a window of $f + 2$ rounds without crashes, there must be a round with the sufficient conditions for a block to be committed for that round.

▶ **Lemma 7.** *In a crash-only execution, let R be a round after GST such that no party crashes between rounds R and $R + f + 2$ (including). There exists a round $R \leq r \leq R + f + 2$ for which there are $2f + 1$ LBR-synchronized(ℓ) invocations with a leader ℓ that is* alive *at round r.*

Proof. First, let us consider the LBR invocations for round R. By Lemma 6, if one honest party returns with a block B with round number R, then $2f + 1$ honest parties return with B, commit it and update *commit_head* accordingly (line 7). In this case, there are $2f + 1$ `choose_leader`$(R+1, B)$ invocations, which all return at line 18. Otherwise, no party return a block with round number R, and thus they all return at line 11. By the code and since a block implies a unique chain, in both cases $2f + 1$ honest parties return the same leader ℓ in `choose_leader`$(R + 1, B)$ (either by reputation or round-robin). By the Pacemaker guarantees and since $R + 1$ occurs after GST, there are at least $2f + 1$ LBR-synchronized(ℓ) invocations. If ℓ is alive at round $R + 1$, we are done. Otherwise, ℓ must have been crashed before round R by the alive definition and lemma assumptions. Thus, by the LBR Blocking property no honest party commits a block for round R and they all choose the same leader for the following round at line 11. The lemma follows by applying the above argument for $R + f + 2 - R + 1 = f + 1$ rounds. ◀

Finally, we bound by $O(f^2)$ the total number of rounds in a crash-only execution for which no honest party commits a block:

▶ **Lemma 8.** *Consider a crash-only execution. After GST, the number of rounds r for which no honest party commits a block formed in r is bounded by $O(f^2)$.*

Proof. Consider a crash-only execution and let $R_1, R_2, \ldots R_k$ the rounds after GST in which parties crash ($k \leq f$). For ease of presentation we call a round for which no honest party commits a block formed in r a *skipped* round. We prove that the number of skipped rounds between R_i and R_{i+1} for $1 \leq i < k$ is bounded. If $R_{i+1} - R_i < f + 4$, then there are at most $f + 4$ rounds and hence at most $f + 4$ skipped rounds. Otherwise, we show that at most $f + 2$ rounds are skipped between rounds R_i and R_{i+1}.

First, by Lemma 7, there exists a round $R_i < R_i + 1 \leq r \leq R_i + 1 + f + 2 < R_{i+1}$ for which there are $2f + 1$ LBR-synchronized(ℓ) invocations with a leader ℓ that is *alive* at round r. By Lemma 5, since no party crashes in round r, $2f + 1$ honest parties return the same leader ℓ' at line 3 of round $r + 1$ and ℓ' is alive at round r. Since no party crashes at round $r + 1$ as well (because $R_{i+1} - R_i \geq f + 4$), ℓ' is alive at round $r + 1$. By the Pacemaker guarantees and since we consider rounds after GST, we conclude that there are at least $2f + 1$ LBR-synchronized(ℓ') invocations for round $r + 1$. By Lemma 5 applied again for round $r + 1$, $2f + 1$ honest parties commit a block for round $r + 1$. Thus, round $r + 1$ is not *skipped*. We repeat the same arguments until round R_{i+1}, and conclude that in each of these rounds a block is committed. Hence, the rounds that can possibly be skipped between R_i and R_{i+1} are $R_i \leq r' < r$. Thus there are $O(f)$ skipped round between R_i and R_{i+1}. For R_k we use similar arguments but since no party crashes after R_k, we apply Lemma 5 indefinitely. We similarly conclude that there are $O(f)$ skipped rounds after R_k. All in all, since $k \leq f$, we get $O(f^2)$ skipped rounds. ◀

We immediately conclude the following:

▶ **Corollary 9.** *Algorithm 1 with Algorithm 2 satisfies Leader-utilization.*

4.1.1 Chain-Quality.

For the purposes of the Chain-quality proof, we say that a block is committed when some honest party commits it. We say that a block B with round number r is *immediately committed* if an honest party commits B in round r. When we refer to a leader elected in of Algorithm 2 from the round-robin mechanism we mean line 11, and when we refer to a leader elected from the reputation mechanism, we mean line 18.

We begin by showing that each round assigned with an honest round-robin leader implies a committed block in that round or the one that precedes it (not necessarily an honest block).

▶ **Lemma 10.** *Let r be a round after GST such that $p_i = (r \mod n)$ is honest. Then, either a Byzantine block with round number $r - 1$ or an honest block with round number $r - 1$ or r is immediately committed.*

Proof. If a block is immediately committed with round number $r - 1$ then we are done. Otherwise, no honest party commits a block with round number $r - 1$ in round $r - 1$, and they all elect the round r leader ℓ using the round-robin mechanism. By the assumption, ℓ is honest.

By the Pacemaker, all honest invocations of $LBR(r, \ell)$ in line 4 are LBR-synchronized(ℓ). Since there are at least $2f + 1$ honest parties, by the LBR Progress property, all honest invocations return the same certified block B with round number r authored by ℓ. Then, the honest parties commit B at line 6. ◀

If there are two consecutive rounds assigned with honest round-robin leaders and in addition the last f committed blocks are Byzantine, then an honest block follows, as proven in the following lemma.

▶ **Lemma 11.** *Let r' be a round after GST such that $p_i = (r' \mod n)$ and $p_j = (r' + 1 \mod n)$ are honest. Suppose f blocks with round numbers in $[r, r')$ with different Byzantine authors are committed. For a block B with round number r' or $r' + 1$ that is immediately committed, there is an honest block with round number $[r, r' + 1]$ on B's implied chain.*

Proof. By the LBR endorsement assumption and property, the author of block B should be either a reputation-based, or a round-robin leader of round r' or $r'+1$. If it is a round-robin leader, then by the lemma assumption, the leader is honest and since B is the head of its implied chain, the proof is complete. Thus, in the following we assume that B's author is a reputation-based leader. By the SMR Agreement property and the lemma assumption, B's implied chain contains f blocks with different Byzantine authors and rounds numbers in $[r, r')$. By the code of the reputation-based mechanism, either all f Byzantine authors are excluded from the *leader_candidates* which implies that B has an honest author, or that there is an honest block with round number in $[r, r')$ on B's implied chain. ◀

Lastly, the following lemma proves that in any window of $5f+2$ rounds an honest block is committed.

▶ **Lemma 12.** *Let r be a round after GST. At least one honest block is committed with a round number in $[r, r+5f+2]$.*

Proof. Suppose for contradiction that no honest block with round number in $[r, r+5f+2]$ is committed. There are at least f rounds r' in $[r, r+3f+1)$, such that rounds $r'-1$ and r' are allocated an honest leader by the round-robin mechanism. By Lemma 10, a block with round number $r'-1$ or r' is immediately committed. Due to Lemma 10 and the contradiction assumption, for any such round r', a Byzantine block with round number $r'-1$ is immediately committed. Since $r'-1$ has an honest round-robin leader, the block must be committed from the reputation mechanism.

It follows that f Byzantine blocks with round numbers in $[r, r+3f+1)$ are immediately committed from the reputation mechanism, and consequently, they all must have different authors. Note that there exists $r' \in [r+3f+1, r+5f+2)$ (in a window of $2f+1$ rounds), such that the round-robin mechanism allocates honest leaders to rounds r' and $r'+1$. By Lemma 10, a block B with round number r' or $r'+1$ is immediately committed. Lemma 11 concludes the proof. ◀

We conclude the following:

▶ **Corollary 13.** *Algorithm 1 with Algorithm 2 satisfies Chain-quality and Liveness.*

Taken jointly, Theorem 9, Theorem 13, and the Agreement property proved in Section 3.3 yield the following theorem:

▶ **Theorem 14.** *Algorithm 1 with Algorithm 2 implements Leader-Aware SMR.*

5 Implementation

We implement Carousel on top of a high-performance open-source implementation of HotStuff[5] [22]. We selected this implementation because it implements a Pacemaker [22], contrarily to the implementation used in the original HotStuff paper[6]. Additionally, it provides well-documented benchmarking scripts to measure performance in various conditions, and it is close to a production system (it provides real networking, cryptography, and

[5] `https://github.com/asonnino/hotstuff`
[6] `https://github.com/hot-stuff/libhotstuff`

persistent storage). It is implemented in Rust, uses Tokio[7] for asynchronous networking, ed25519-dalek[8] for elliptic curve based signatures, and data-structures are persisted using RocksDB[9]. It uses TCP to achieve reliable point-to-point channels, necessary to correctly implement the distributed system abstractions. By default, this HotStuff implementation uses traditional round-robin to elect leaders; we modify its `LeaderElector` module to use Carousel instead. Implementing our mechanism requires adding less than 200 LOC, and does not require any extra protocol message or cryptographic tool. We are open-sourcing Carousel[10] along with any measurements data to enable reproducible results[11].

6 Evaluation

We evaluate the throughput and latency of HotStuff equipped Carousel through experiments on Amazon Web Services (AWS). We then show how it improves over the baseline round-robin leader-rotation mechanism. We particularly aim to demonstrate that Carousel (i) introduces no noticeable performance overhead when the protocol runs in ideal conditions (that is, all parties are honest) and with a small number of parties, and (ii) drastically improves both latency and throughput in the presence of crash-faults. Note that evaluating BFT protocols in the presence of Byzantine faults is still an open research question [2].

We deploy a testbed on AWS, using `m5.8xlarge` instances across 5 different AWS regions: N. Virginia (us-east-1), N. California (us-west-1), Sydney (ap-southeast-2), Stockholm (eu-north-1), and Tokyo (ap-northeast-1). Parties are distributed across those regions as equally as possible. Each machine provides 10Gbps of bandwidth, 32 virtual CPUs (16 physical core) on a 2.5GHz, Intel Xeon Platinum 8175, 128GB memory, and run Linux Ubuntu server 20.04.

In the following sections, each measurement in the graphs is the average of 5 independent runs, and the error bars represent one standard deviation. Our baseline experiment parameters are 10 honest parties, a block size of 500KB, a transaction size of 512B, and one benchmark client per party submitting transactions at a fixed rate for a duration of 5 minutes. We then crash and vary the number of parties through our experiments to illustrate their impact on performance. The leader timeout value is set to 5 seconds for runs with 10 and 20 parties and increased to 10 seconds for runs with 50 parties. When referring to *latency*, we mean the time elapsed from when the client submits the transaction to when the transaction is committed by one party. We measure it by tracking sample transactions throughout the system.

6.1 Benchmark in Ideal Conditions

Figure 1 depicts the performance of HotStuff with both Carousel and the baseline round-robin running with 10, 20, and 50 honest parties. For runs with a small number of parties (e.g., 10), the performance of the baseline round-robin HotStuff is similar to HotStuff equipped with Carousel. We observe a peak throughput around 70,000 tx/s with a latency of around 2 seconds. This illustrates that the extra code required to implement Carousel has negligible overhead and does not degrade performance when the total number of parties is small. When

[7] `https://tokio.rs`
[8] `https://github.com/dalek-cryptography/ed25519-dalek`
[9] `https://rocksdb.org`
[10] Link omitted for blind review.
[11] Link omitted for blind review.

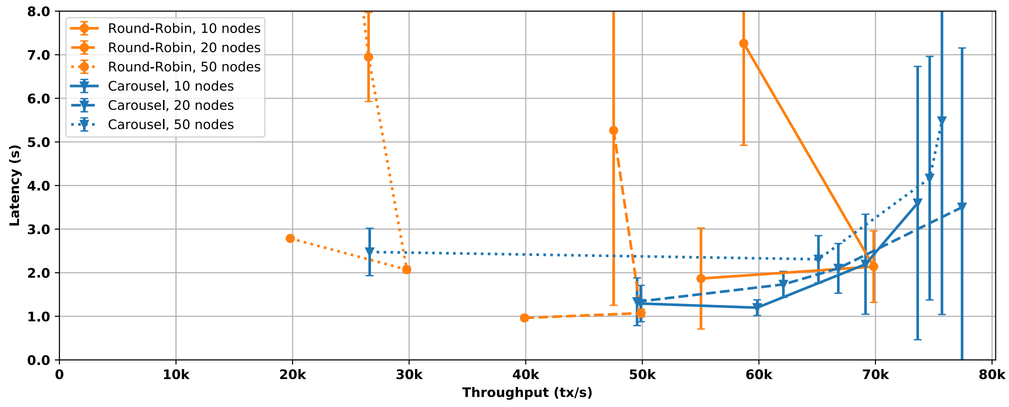

Figure 1 Comparative throughput-latency performance of HotStuff equipped with Carousel and with the baseline round-robin. WAN measurements with 10, 20, 50 parties. No party faults, 500KB maximum block size and 512B transaction size.

increasing the system's size (to 20 and 50 parties), HotStuff with Carousel greatly outperforms the baseline: the bigger the system's size, the bigger the performance improvement. With 50 nodes, the throughput of our mechanism-based HotStuff increases by over 2x with respect to the baseline, and remains comparable to the 10-parties testbed. After a few initial timeouts, Carousel has the benefit to focus on electing performant leaders. Leaders on more remote geo-locations that are typically slower are elected less often, the protocol is thus driven by the most performant parties. Similar ideas were presented in [18] in the context of distributed data storage, where a leader placement was optimized based on replicas' locations. In our experiments, latency is similar for both implementations and around 2-3 seconds.

6.2 Performance under Faults

Figure 2 depicts the performance of HotStuff with both Carousel and the baseline round-robin when a set of 10 parties suffers 1 or 3 crash-faults (the maximum that can be tolerated). The baseline round-robin HotStuff suffers a massive degradation in throughput as well as a dramatic increase in latency. For three faults, the throughput of the baseline HotStuff drops over 30x and its latency increases 5x compared to no faults. In contrast, HotStuff equipped with Carousel maintains a good level of throughput: our mechanism does not elect crashed leaders, the protocol continues to operate electing leaders from the remaining active parties, and is not overly affected by the faulty ones. The reduction in throughput is in great part due to losing the capacity of faulty parties. When operating with 3 faults, Carousel provides a 20x throughput increase and about 5x latency reduction with respect to the baseline round-robin.

Figure 3 depicts the evolution of the performance of HotStuff with both Carousel and the baseline round-robin when gradually crashing nodes through time. For roughly the first minute, all parties are honest; we then crash 1 party (roughly) every minute until a maximum of 3 parties are crashed. The input transaction rate is fixed to 10,000 tx/s throughout the experiment. Each data point is the average over intervals of 10 seconds. For roughly the first minute (when all parties are honest), both systems perform ideally, timely committing all input transactions. Then, as expected, the baseline round-robin HotStuff suffers from temporary throughput losses when a crashed leader is elected. Similarly, its latency increases with the number of faulty parties and presents periods where no transactions are committed at all. In contrast, HotStuff equipped with Carousel delivers a stable throughput by quickly

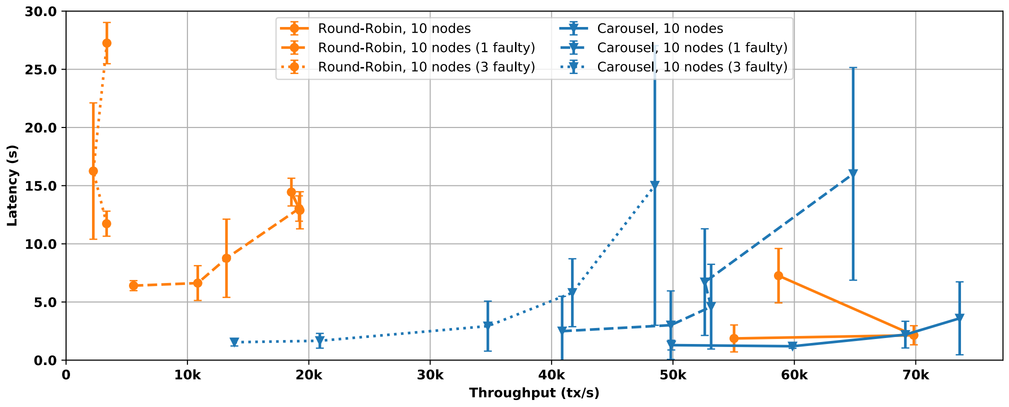

Figure 2 Comparative throughput-latency performance of HotStuff equipped with Carousel and with the baseline round-robin. WAN measurements with 10 parties. Zero, one and three party faults, 500KB maximum block size and 512B transaction size.

detecting and eliminating crashed leaders. Its latency is barely affected by the faulty parties. This graph clearly illustrates how Carousel allows HotStuff to deliver a seamless client experience even in the presence of faults.

7 Conclusions

Leader-rotations mechanisms in chaining-based SMR protocols were previously overlooked. Existing approaches degraded performance by keep electing faulty leaders in crash-only executions. We captured the practical requirement of leader-rotation mechanism via a Leader-utilization property, use it define the Leader-Aware SMR problem, and described an algorithm that implements it. That is, we presented a locally executed algorithm to rotate leaders that achieves both: Leader-utilization in crash-only executions and Chain-quality in Byzantine ones. We evaluated our mechanism in a Hotstuff-based open source system and demonstrated drastic performance improvements in both throughput and latency compared to the round-robin baseline.

References

1 Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.

2 Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, Zekun Li, Avery Ching, and Dahlia Malkhi. Twins: Bft systems made robust. In *25th International Conference on Principles of Distributed Systems (OPODIS 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

3 Dan Boneh, Manu Drijvers, and Gregory Neven. The modified BLS multi-signature construction. `https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html`, 2018.

4 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

5 Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, 2016.

6 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget.

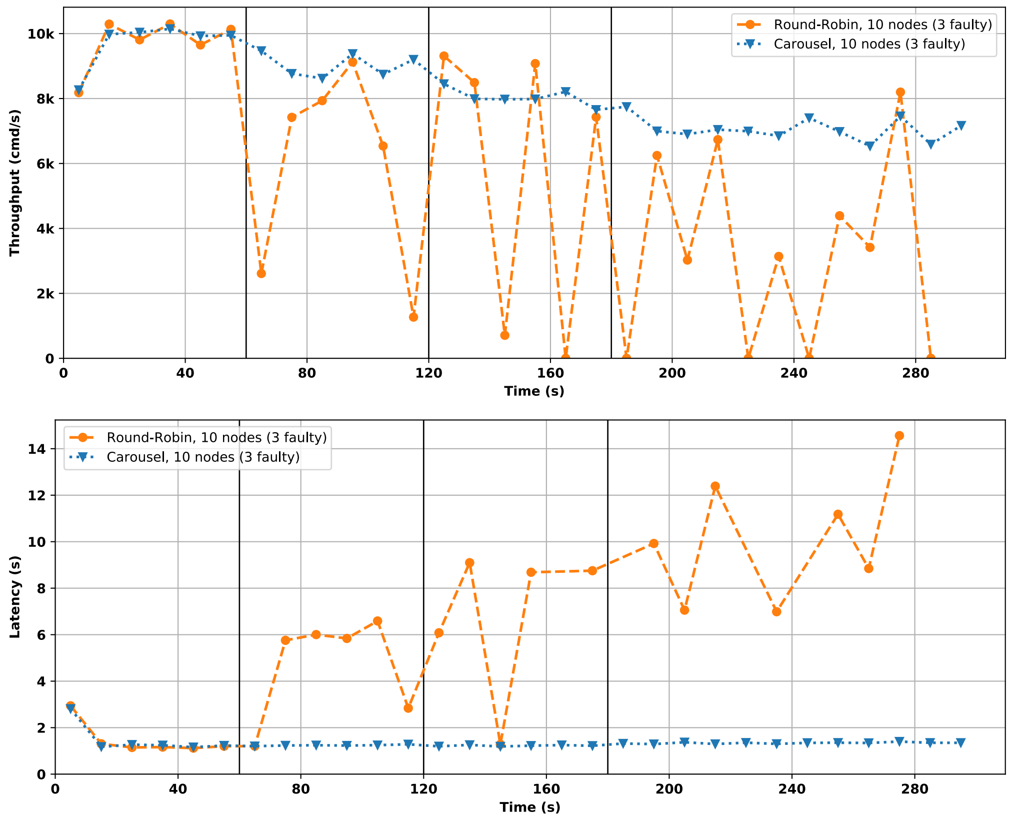

Figure 3 Comparative performance of HotStuff equipped with Carousel and with the baseline round-robin when gradually crashing nodes through time. The input transactions rate is fixed to 10,000 tx/s; 1 party (up to a maximum of 3) crashes roughly every minute. WAN measurements with 10 parties, 500KB maximum block size and 512B transaction size.

7 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

8 Benjamin Y Chan and Elaine Shi. Streamlet: Textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.

9 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

10 Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.

11 Mahimna Kelkar, Fan Zhang, Steven Goldfeder, and Ari Juels. Order-fairness for byzantine consensus. In *Annual International Cryptology Conference*, pages 451–480. Springer, 2020.

12 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Communications of the ACM*, volume 21, page 558–565. 1978.

13 Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

14 Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. {XFT}: Practical fault tolerance beyond crashes. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 485–500, 2016.

15 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine View Synchronization. *Cryptoeconomic Systems*, 1(2), oct 22 2021.

16 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

17 Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

18 Artyom Sharov and Alexander Shraer Arif Merchant Murray Stokely. Take me to your leader! online optimization of distributed storage configurations. *Proceedings of the VLDB Endowment*, 8(12), 2015.

19 Alexander Spiegelman. In search for an optimal authenticated byzantine agreement. In *35th International Symposium on Distributed Computing*, 2021.

20 Alexander Spiegelman, Arik Rinberg, and Dahlia Malkhi. Ace: Abstract consensus encapsulation for liveness boosting of state machine replication. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

21 The Diem Team. Diembft v4: State machine replication in the diem blockchain. `https://developers.diem.com/docs/technical-papers/state-machine-replication-paper.html`.

22 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: Bft consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

Appendix A Correctness

▶ **Lemma 15.** *If* `choose_leader` *returns the same honest party at all honest parties for infinitely many rounds, then each honest party commits an unbounded number of blocks.*

Proof. If `choose_leader` returns the same honest party at all honest parties for infinitely many rounds, then there are infinitely many rounds after GST for which it does so. Let r be such a round. By the Pacemaker guarantees, all honest parties make LBR-synchronized(ℓ) invocations with the same honest leader ℓ returned from the `choose_leader` procedure. By the LBR Progress property, they all return a certified block B and commit it at line 6. ◀

▶ **Lemma 16.** *In a crash-only execution, let r be a round with $k \geq 2f+1$ LBR-synchronized(ℓ) invocations, such that ℓ is alive at round r, then these k invocations return a certified B with round number r authored by ℓ.*

Proof. Let π_1 be a crash-only execution, such that round r has $k \geq 2f+1$ *LBR-synchronized(ℓ) invocations* with a leader ℓ that is alive at round r. If ℓ is honest, then the LBR Progress property concludes the proof.

Otherwise, ℓ is faulty and by definition it crashes in round $> r$. Let π_2 be a crash-only execution that is identical to π_1 until ℓ crashes, and the rest of π_2 is an arbitrary execution where the honest parties in π_1 remain honest but ℓ never crashes and is also honest. Thus, in π_2 the preconditions of the LBR Progress property hold and all k *LBR-synchronized(ℓ) invocations* return a certified B with round number r authored by ℓ.

An $LBR(r, \ell)$ invocation by any party p completes within Δ_l time, and starts immediately after Pacemaker's `new_round(r)` notification at p (because `choose_leader` is computed locally and takes 0 time). By Pacemaker's guarantees, no party receives `new_round`$(r+1)$ notification until $\Delta_p = \Delta_l$ time after the last `new_round`$(r+1)$ notification at some party, hence all $LBR(r, \ell)$ invocations must complete before any party receives a `new_round`$(r+1)$ notification.

π_1 and π_2 are identical until ℓ crashes, which must happen after ℓ receives its `new_round`$(r+1)$ notification from the Pacemaker. This is because ℓ is alive in round r and follows the protocol, invoking LBR in round $r+1$ after receiving the `new_round`$(r+1)$ notification. As a result, π_1 and π_2 are indistinguishable to all $LBR(r, \ell)$ invocations, and the k *LBR-synchronized(ℓ) invocations* in π_1 return certified block B with round number r authored by ℓ as in π_2, as desired. ◀

2.4 Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words

Appears in the 26th International Conference on Principles of Distributed Systems (OPODIS 2022).

Make Every Word Count: Adaptive Byzantine Agreement with Fewer Words

Shir Cohen
Technion, Israel

Idit Keidar
Technion, Israel

Alexander Spiegelman
Aptos, USA

Abstract

Byzantine Agreement (BA) is a key component in many distributed systems. While Dolev and Reischuk have proven a long time ago that quadratic communication complexity is necessary for worst-case runs, the question of what can be done in practically common runs with fewer failures remained open. In this paper we present the first Byzantine Broadcast algorithm with $O(n(f+1))$ communication complexity in a model with resilience of $n = 2t+1$, where $0 \leq f \leq t$ is the actual number of process failures in a run. And for BA with strong unanimity, we present the first optimal-resilience algorithm that has linear communication complexity in the failure-free case and a quadratic cost otherwise.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine Agreement, Byzantine Broadcast, Adaptive communication

Funding *Shir Cohen*: Supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.

1 Introduction

Byzantine Agreement (BA) is a key component in many distributed systems. As these systems are being used at larger scales, there is an increased need to find efficient solutions for BA. Arguably, the most important aspect of an efficient BA solution is its communication costs. That is, how much information needs to be transferred in the network to solve the BA problem. Indeed, improving the communication complexity, often measured as word complexity, was the focus of many recent works and deployed systems [1, 11, 16, 2, 13, 7].

In the BA problem, a set of n processes attempt to agree on a *decision* value despite the presence of Byzantine processes. One of the properties of a BA algorithm is a threshold t on how many Byzantine processes it can withstand. Namely, the algorithm is correct as long as up to t processes are corrupted in the course of a run. In this paper we focus on $n = 2t+1$ and we assume a trusted setup of a public-key infrastructure (PKI) that enables us to use a threshold signature scheme [15, 4, 6].

A large and growing body of literature has investigated how to reduce the word complexity of BA algorithms. Recently, Momose and Ren [13] have presented a synchronous protocol with $O(n^2)$ words, which meets Dolev and Reischuk's long-standing lower bound [9]. Spiegelman [16] considered the more common case, where the number of actual failures, denoted by f, is smaller than t with resilience of $n = 3t+1$. In this paper we consider better resilience and ask:

Can we design a BA protocol with $O(n(f+1))$ communication complexity in runs with $f \leq t$ failures, where $n = 2t+1$?

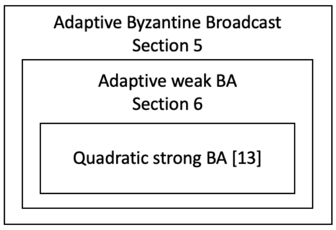

Figure 1 Relation between various Byzantine Agreement solutions. Each box uses the primitives within it.

Whereas Dolev and Reischuk's better-known lower bound applies to worst-case runs, they further proved a lower bound of $\Omega(nt)$ signatures in failure-free runs ($f = 0$) in a model with a PKI. At the time, one could have thought that this bound extends to the communication complexity, rendering it $\Omega(nt)$ even with small f values. However, the introduction of threshold signature schemes [8, 15, 4, 6] exposed the possibility to compact many signatures into one word, potentially saving many words.

In this paper, we first revisit the original problem as stated in Dolev and Reischuk's work. In this problem there is a single sender who proposes a value and we refer to this problem as Byzantine Broadcast (BB). We prove that although $O(nt)$ signatures are inevitable, $O(nt)$ messages are not necessary with $f \in o(t)$ failures by presenting an adaptive BB solution with $O(n(f + 1))$ words.

The idea behind our algorithm is to reduce this problem to another BA variant. There is a simple reduction from BB to BA with the strong unanimity validity property (from hereon: *strong BA*), which states that if all correct processes propose the same value, this is the only allowed decision. In this reduction, the sender initially sends its value to all other processes who then run a BA solution. Unfortunately, no *adaptive* strong BA is known to date. I.e., a strong BA solution where communication complexity depends on f, rather than on t. Instead, in Section 5 we reduce the problem to a new *weak BA* problem with a weaker validity property, *unique validity*, which we define in this paper.

Intuitively, the validity condition of weak BA is somewhere between weak unanimity, where if all processes are correct and propose the same value this is the only allowed decision, and external validity [5], where a decision value must satisfy some external predicate. In weak BA, one can define its desired predicate and the requirement is that if all correct processes propose the same value and Byzantine processes cannot devise a value that satisfies the chosen predicate, then the decision must be valid. Otherwise, $\perp$ is allowed.

While the unique validity condition seems to be weak, it is surprisingly powerful when provided the "right" external predicate. For example, we can determine that a value is valid if it has at least $t + 1$ unique signatures, assuring that some correct process in the system knows this value. Unique validity may be of independent interest as a tool for designing algorithms. We present our adaptive weak BA in Section 6. The weak BA, in turn, exploits the quadratic solution by Momose and Ren [13]. Figure 1 describes the relation between the various solutions.

Finally, we consider strong BA. In Section 7, we present the first optimally resilient strong binary BA protocol with $O(n)$ communication complexity in the failure-free case. This leaves open the question whether a fully adaptive (to any f) strong BA protocol exists. We summarize the results in Table 1.

Table 1 Bounds on communication complexity of deterministic synchronous Byzantine Agreement algorithms with resilience $n = 2t + 1$.

	Upper Bound	**Lower Bound**
Byzantine Broadcast	$O(n(f+1))$ **Section 5 + Section 6**	$\Omega(nf)$ ($\Omega(n^2)$ signatures) [9]
Strong BA	$O(n^2)$ multi-valued Momose-Ren [13] $O(n)$ with $f = 0$, binary **Section 7**	$\Omega(nf)$ binary ($\Omega(n^2)$ signatures) [9]
Weak BA	$O(n(f+1))$ multi-valued **Section 6**	$\Omega(n)$

2 Model and Preliminaries

We consider a distributed system consisting of a well-known static set Π of n processes and an adaptive adversary. The adversary may adaptively corrupt up to $t < n, n = 2t + 1$ processes in the course of a run. A corrupted process is *Byzantine*; it may deviate arbitrarily from the protocol. In particular, it may crash, fail to send or receive messages, and send arbitrary messages. As long as a process is not corrupted by the adversary, it is *correct* and follows the protocol. We denote by $0 \le f \le t$ the actual number of corrupted processes in a run.

Cryptographic tools. We assume a trusted public-key infrastructure (PKI) and a computationally bounded adversary. Hence, we can construct and use a threshold signature scheme [15, 4, 6]. We denote by $\langle m \rangle_p$ the message m signed by process p. Using a (k, n)-threshold signature scheme, k unique signatures on the same message m can be batched into a threshold signature for m with the same length as an individual signature. For simplicity we abstract away the details of cryptography and assume the threshold signature schemes are ideal. In practice, our results hold except with arbitrarily small probability, depending on the security parameters.

Communication. Every pair of processes is connected via a reliable link. If a correct process p_i receives a message m indicating that m was sent by a correct process p_j, then m was indeed generated by p_j and sent to p_i. The network is synchronous. Namely, there is a known bound δ on message delays, allowing us to design protocols that proceed in rounds. Specifically, if a correct process sends a message to any other correct process at the beginning of some round, it is received by the end of the same round.

Complexity. We use the following standard complexity notions [2, 16, 13]. While measuring complexity, we say that a *word* contains a constant number of signatures and values from a finite domain, and each message contains at least 1 word. The communication complexity of a protocol is the maximum number of words sent by all correct processes, across all runs. The *adaptive* complexity is a complexity that depends on f.

3 Problem Definitions

We consider a family of agreement problems all satisfy agreement and termination defined as follows:

Agreement No two correct processes decide differently.
Termination Every correct process eventually decides.

In addition, each variant of the problem satisfies some validity property. In the Byzantine Broadcast (BB) problem, a designated sender has an input to broadcast to all n processes. The goal is that all correct processes decide upon the sender's value. If the sender is Byzantine, however, it is enough that all correct processes decide upon some common value. Formally,

▶ **Definition 1** (Byzantine Broadcast). *In Byzantine Broadcast, a designated sender* sender *has an input value v_{sender} to broadcast to all processes, and each correct process decides on an output value $decision_i$. BB solution must satisfy agreement, termination and the following validity property:*

Validity *If* sender *is correct, then all correct processes decide v_{sender}.*

Byzantine Agreement (BA) is a closely related problem to BB. In this problem, a set Π of n processes each propose an initial value and they all attempt to reach a common decision. In addition, the decided value must be "valid" in some sense that makes the problem non-trivial. The classic notion of validity states that if all correct processes in Π share the same initial value, then the decision must be on this value. This property is known as *strong unanimity*, and it entails a limitation on the resilience of a protocol, requiring that $n \geq 2t + 1$. For hereon we refer to BA with strong unanimity validity condition as *strong BA*. Formally,

▶ **Definition 2** (Strong Byzantine Agreement). *In Byzantine Agreement, each correct process $p_i \in \Pi$ proposes an input value v_i and decides on an output value $decision_i$. Any strong BA solution must satisfy agreement, termination and the following validity property:*

Strong unanimity *If all correct processes propose the same value v, then the output is v.*

A different validity property requires that a decision satisfies some external boolean predicate (we call such value a *valid* value). It is used under the assumption that all correct processes propose valid values. This is known as *external validity* [5] and only requires $n > t$. External validity by itself is trivial in case there is a well-known predefined value that satisfies the predicate. However, it is commonly used in settings with signatures, where valid values can be verified by all but generated only by specific users or sets thereof. For instance, consider a predicate that verifies that v is signed by $n - t$ processes – no process can unilaterally generate a default valid value.

Our notion of unique validity adopts external validity to allow default values to be decided in cases when there is no unanimous valid value. We say that a value v *exists* in a run of a BA protocol if v is either the input value of a correct process or can be generated by a Byzantine process. E.g., any value signed by a non-Byzantine process cannot be generated locally by a Byzantine process. Unique validity stipulates that there is a default value if and only if there exists more than one valid value in a BA run. Formally,

▶ **Definition 3** (Weak Byzantine Agreement). *In weak Byzantine Agreement, each correct process $p_i \in \Pi$ proposes an input value v_i and decides on an output value $decision_i$. Any weak BA solution must satisfy agreement, termination and the following validity property:*

Unique Validity *Assume an arbitrary predicate $validate(v) \in \{true, false\}$ that can be computed locally. If a correct process decides v then either $v = \perp$ or $validate(v) = true$, and if $v = \perp$ then more than one valid value exists in the run.*

As the definition suggests, unique validity is satisfied in weak BA with respect to any chosen external predicate. This allows for the application level to determine the desired properties, and choose the relevant external predicate accordingly. As a simple example, one can think of a predicate that specifies that a value is valid if it is signed by at least $t + 1$ processes stating that this value was their initial value. In this scenario, unique validity yields exactly the common strong unanimity property on the underlying signed values.

In fact, unique validity is a useful tool when designing distributed algorithms as it allows to use BA as a framework. Different applications may require different validity conditions, yet still unique validity prevents the system from having a trivial solution in the presence of Byzantine processes. Note, in addition, that every solution to BA with external validity property immediately solves weak BA.

4 Related Work

The starting point of this work goes back to 1985 when Dolev and Reischuk proved two significant lower bounds for the Byzantine Broadcast problem. Specifically, they have studied the worst-case message complexity over all runs and proved it to be $\Omega(nt)$. Moreover, in the authenticated model, which was somewhat undeveloped at the time, they proved a lower bound of $\Omega(nt)$ signatures – even in a failure-free run.

Since the publication of their fundamental results, the paradigm of complexity measurement has shifted. The number of messages is of little importance nowadays, compared to the number of words it entails. The total number of words (the communication complexity) better reflects the load on the system and is commonly used today when analyzing distributed algorithms. For example, Dolev and Reischuk presented in their paper a BB algorithm that matches their messages' lower bound. It requires $O(nt)$ messages, but as a single message can be composed of many different signatures it requires a cubic number of words. It was not until recently that a solution with quadratic communication complexity was presented for synchronous BA with optimal resilience [13].

Dolev and Reischuk's complementary lower bound on signatures does not translate to a bound on the communication complexity of an algorithm. Only a few years after Dolev and Reischuk's work, the threshold signature scheme was introduced [8]. This scheme allows multiple signatures to be compacted into a single combined signature of the same size. That is, a single word can carry multiple signatures. In this work, we focus on the communication complexity of the BB and BA problems while taking advantage of such schemes.

To make our algorithms efficient in real-world systems, we adjust the complexity to match the actual number of faults. Moreover, we do so without compromising the worst-case complexity. If all t possibly Byzantine processes crash, the complexity of our algorithms is $O(nt)$. However, in most runs, where systems do not exhibit the worst crash patterns, the complexity is much lower. In fact, it is linear in the number of faults times n.

While consensus algorithms were designed to be adaptive in the number of failures over 30 years ago [10], these works focus on the number of rounds that it takes to reach a decision rather than on communication complexity. A special case of adaptivity is focusing on failure-free runs. This problem was addressed both by Amdur et al. [3] (only for crash failures) and by Hadzilacos and Halpern [12]. However, both works measure the number of messages rather than words and have sub-optimal communication complexity.

A recent work by Spiegelman [16] tackled the problem of adaptive communication complexity in the asynchronous model. It presents a protocol that achieves correctness in asynchronous runs and requires $O(ft + t)$ communication in synchronous runs. However, due to the need to tolerate asynchrony, its resilience is only $n \geq 3t + 1$. This solution relies on threshold signatures schemes, as we do.

As noted also by Momose and Ren [13], designing optimally-resilient protocols for the synchronous model limits the use of threshold signatures. While this primitive has been used in various eventually synchronous and asynchronous works over the last few years [1, 16, 14, 6], usually with a threshold of $n-t$. Using this threshold in settings with resilience $n = 3t+1$, we get certificates signed by at least $t+1$ correct processes. However, for a resilience of $n = 2t+1$, this is no longer the case. The threshold signatures "lose" their power as $n - t = t + 1$ for which no intersection properties between correct processes signing two distinct certificates can be derived. In this work, we exploit threshold signatures with this improved resilience by carefully choosing a better threshold for our needs, as we discuss in Section 6. We mention that although not using threshold schemes, Xiang et al. [17] also benefit from collecting more than $n - t$ signatures in some scenarios.

5 From Weak BA to Adaptive Byzantine Broadcast

In this section we study the BB problem, and optimize its adaptive communication complexity over all runs. We present a new BB protocol with resilience $n = 2t + 1$ and adaptive communication complexity of $O(n(f + 1))$.

Recall that in the BB problem there is only one sender who aims to broadcast its initial value and have all correct processes agree on it. If the sender is Byzantine, it may attempt to cause disagreement across correct processes. There is a known simple and efficient reduction from BB to strong BA. Given a strong BA solution, the designated sender starts by sending its value to all processes, and then they all execute the BA solution and decide on its output. It is easy to see that if the sender is correct, all correct processes begin the strong BA algorithm with the same input, and by strong unanimity they then decide upon the sender's value.

However, trying to apply the same reduction from BB to weak BA no longer works. If the sender is Byzantine, the correct processes do not have a valid initial value for the BA. Nonetheless, in this section we present a reduction from BB to weak BA[1], which incurs a cost of $O(n(f + 1))$ words. Thus, together with an adaptive weak BA with the same complexity, we obtain a synchronous adaptive BB algorithm with a total of $O(n(f + 1))$ words and resilience $n = 2t + 1$. At this point we assume that such adaptive weak BA is given as a black box. An implementation for this primitive is presented in Section 6.

Algorithm 1 BB algorithm: code for process p_i, *sender*'s input is v_{senedr}

```
Initially $v_i$, $val$, $decision$, $ba\_decision = \bot$

Round 1:
1: if $sender = p_i$ then
2:     send $\langle v_{sender} \rangle_{sender}$ to all
3: if received message $\langle v \rangle_{sender}$ from $sender$ then
4:     $v_i \leftarrow \langle v \rangle_{sender}$
5: for $j = 1$ to $n$ do
6:     $val \leftarrow invokePhase(j, v_i)$
7:     if $val \neq \bot$ then
8:         $v_i \leftarrow val$
9: $ba\_decision \leftarrow$ weak BA with $BB\_valid$ predicate and initial value $v_i$
10: if $ba\_decision$ is of the form $\langle v \rangle_{sender}$ then
11:     $decision \leftarrow v$
12: else
13:     $decision \leftarrow \bot$
```

[1] This reduction only works if $n \geq 2t + 1$.

Our algorithm, presented in Algorithms 1 and 2, is composed of three parts. The first part (lines 1 – 4 in Algorithm 1) is the first round in which the leader disseminates its value. Processes that receive that value adopt it as their BA initial value (line 4). The second part (lines 5 – 8 in Algorithm 1 and Algorithm 2) is a "vetting" part. It consists of n phases, with a rotating leader. Leaders initiate phases to learn about the first part's initial value. Finally, the third part (lines lines 9 – 13 in Algorithm 1) is a weak BA execution.

Deciding upon the weak BA output takes care of the agreement and termination properties. It is left to (1) satisfy the BB validity property and (2) make sure that the preconditions for the weak BA hold, that is, each correct process has a valid input to propose. To achieve these properties, we define the $BB\_valid(v)$ predicate in the following way. $BB\_valid(v) = true$ if and only if v is signed by either the sender or by $t + 1$ processes.

Note that if the sender happens to be Byzantine, it is acceptable to decide on any value. However, it is important to make sure that if the sender is correct, then the only valid value is its initial BB input. Simply setting a value to be valid only if it is signed by the sender would not work, as it allows a faulty sender to cause a scenario in which there are no valid values to agree upon by not sending its value to any process. Note that we cannot simply fix this by introducing some default valid value: If we were to do so, it would be valid to agree on that value also in the case of a correct sender, violating the BB validity condition.

Our algorithm makes sure that if the sender is correct, the second condition in the *BB_valid* definition cannot be satisfied, and hence there is only one possible outcome to the BA algorithm. However, if the sender is Byzantine, it is guaranteed that there is some value to decide upon. That is, all correct processes start the weak BA with an initial value that satisfies the predicate.

In the vetting part of the algorithm, we ensure that the above-mentioned conditions hold. Moreover, we do so with a communication complexity that is adaptive to the number of actual process failures. The core idea is to work in leader-based *phases*. Every phase has a unique leader and is composed of a constant number of leader-to-all and all-to-leader synchronous rounds. Every phase is initiated by a leader-to-all message. If the leader decides not to send the initial message then no messages will be sent by correct processes in this phase and we say that this phase is *silent*, and otherwise, it is *non-silent*. In our algorithm, a phase is non-silent if the phase's leader did not choose an initial value for the BA prior to that phase.

In every phase, each process p_i starts the phase with some initial value v_i and if the phase is non-silent it returns some value. The requirements from the phase are: (1) If the phase's leader is correct and the phase is non-silent, then all correct processes return a valid value. (2) All correct processes return either $\perp$ or a valid v. And (3) if the sender is correct, then no correct process returns a value signed by $t + 1$ processes.

Upon a non-silent phase, the leader starts by asking all processes for help by sending a `help_req` message (line 16). A correct process that receives a help request message answers the leader. If it has set a BA initial value, it sends it to the leader at line 19, and otherwise, it sends a signed `idk` (i don't know) message at line 21. If the leader receives a value signed by the designated sender it broadcasts it (line 24). Otherwise, if it receives $t+1$ `idk` messages, it uses a threshold signature scheme to create an `idk` quorum certificate and broadcasts it (line 27). A process that receives from the leader a value signed by either the sender or any $t + 1$ processes returns it. Otherwise, it returns $\perp$.

At the end of each non-silent phase, a correct process that returns a $v \neq \perp$ from the phase, updates its local v_i accordingly at line 8. This value at the end of the n^{th} phase is the input for the weak BA algorithm. Since we execute n phases, all correct processes set valid values by the end of all phases. This is because once there is a correct process that did not set a value it initiates its phase and then all correct processes return with a valid value. At this point, all processes execute the weak BA and decide upon its output (line 9).

Algorithm 2 *invokePhase*(j, v_i): code for process p_i

```
14: $leader \leftarrow p_{j \bmod n}$
    Round 1:
15: if $leader = p_i$ and $v_i = \perp$ then
16:     broadcast the message $\langle \mathsf{help\_req}, j\rangle_{leader}$
    Round 2:
17: if received $\langle \mathsf{help\_req}, j\rangle_{leader}$ then
18:     if $v_i \neq \perp$ then
19:         send $\langle v_i, j\rangle$ to $leader$
20:     else
21:         send $\langle \mathsf{idk}, j\rangle_{p_i}$ to $leader$
    Round 3:
22: if $leader = p_i$ then
23:     if received $\langle v', j\rangle$ s.t. $v' = \langle v\rangle_{sender}$ then
24:         broadcast the message $\langle \langle v\rangle_{sender}, j\rangle$
25:     else if received $t+1$ unique signatures $\langle \mathsf{idk}, j\rangle_{p'}$ then
26:         batch these messages into $QC_{\mathsf{idk}}$ using a $(t+1, n)$-threshold signature scheme
27:         broadcast the message $\langle QC_{\mathsf{idk}}, j\rangle$
28: if received $\langle v, j\rangle$ from $leader$ and $BB\_valid(v) = true$ then
29:     return $v$
30: else
31:     return $\perp$
```

5.1 Correctness

We start by proving the phase's requirements. First, immediately from lines 29 – 31 we get that all correct processes return either $\perp$ or a valid v. Next, the following lemma shows that in non-silent phases with correct leaders all correct processes return a valid value.

▶ **Lemma 4.** *If a phase is non-silent and its leader is correct, then all correct processes return a valid value.*

Proof. If the leader is correct it broadcasts a help_req message at line 16. All correct processes then answer at round 2. If the leader receives a value signed by the sender at line 23, it broadcasts it at line 24. Otherwise, no correct processes received a value signed by the sender and sends an idk message at line 21. Since $n = 2t + 1$, the leader receives at least $t + 1$ idk messages (from the correct processes) and forms an idk certificate. It broadcasts this value at line 27. In both cases, all correct processes return a valid value at line 29. ◀

The next lemma proves that if all correct processes invoke a phase with a value other than $\perp$, then they can return only one type of a valid value – a value signed by the sender.

▶ **Lemma 5.** *If all correct processes invoke a phase with value $v \neq \perp$, there does not exists a value signed by $t + 1$ processes in the system.*

Proof. If all correct processes invoke a phase with value $v \neq \bot$, they reply to the help_req messages at line 19 and never send an idk message. Since there are at most t Byzantine processes, the leader cannot receive $t+1$ idk messages and form an idk certificate signed by $t+1$ different processes. ◀

We now prove the correctness of the BB algorithm. First, to be able to use the weak BA, all correct processes must execute it with valid initial values.

▶ **Lemma 6.** *All correct processes execute line 9 with a valid initial value.*

Proof. Let p_i be a correct process. In n phases, there is one phase with p_i as leader. If p_i has updated v_i prior to that phase, it happened either line 4 or at line 8. Immediately from the code we get that in both cases p_i updates a valid value. If p_i did not update a value, it initiates a non-silent phase, and by Lemma 4 returns a valid value. ◀

Note that agreement and termination stem immediately from the code and the correctness of the weak BA. The following lemma proves validity.

▶ **Lemma 7.** *If* sender *is correct, then all correct processes decide* v_{sender}.

Proof. If *sender* is correct then all correct process learn v_{sender} by the end of round 1 and update their values at line 4. By Lemma 5, in no phase can any process create a value signed by $t+1$ processes. Thus, when executing the weak BA v_{sender} signed by the sender is the only valid value that exists in the run. By unique validity and since the *sender* does not sign more than one initial value, v_{sender} is the only possible BA output. It follows that all correct processes execute line 11 and return the sender's value. ◀

We conclude the following theorem:

▶ **Theorem 8.** *Algorithm 1 solves BB.*

5.2 Complexity

We prove that the complexity of Algorithms 1 and 2 is $O(n(f+1))$.

Each non-silent phase is composed of a constant number of all–to–leader and leader–to–all rounds and thanks to the use of threshold signatures, all messages sent have a size of one word. Thus, each phase incurs $O(n)$ words. In total, there are potentially n phases. However, it follows from Lemma 4 that after the first non-silent phase by a correct leader, all the following phases with correct leaders are silent. Thus, the number of non-silent phases is linear in f. We conclude that all phases in lines 5 – 8 use $O(n(f+1))$ words. The complexity of the weak BA black box is also $O(n(f+1))$ (as we will show in the next section), resulting in a total of $O(n(f+1))$ words.

6 Adaptive Weak BA

In this section, we present a synchronous adaptive weak BA algorithm with resilience $n = 2t+1$. This algorithm is the missing link for the adaptive BB presented in the previous section. Once again, we use the concept of phases and exploit the pattern of possible *silent* phases. In this algorithm, the phases are slightly different and the decision to start a phase as a leader depends on whether or not the leader has reached a decision in previous phases.

Unlike the BB problem, in BA every process begins the algorithm with its own input value. Communication-efficient solutions to this problem usually employ threshold signatures schemes [1, 16]. This technique is widely used in asynchronous and eventually synchronous protocols, with resilience $n = 3t+1$. In these contexts, one can use a scheme of $(n-t)$-out-of-n signatures and benefit from the fact that any two such quorum certificates intersect by at least $t+1$ processes, and therefore at least one correct process.

Unfortunately, when trying to apply the same technique to a system with resilience $n = 2t+1$, it fails. A correct process might be unable to obtain $2t+1$ unique signatures on any value as Byzantine processes might not sign it. On the other hand, a quorum certificate with only $t+1$ unique signatures is not very useful as it does not guarantee the desired intersection property.

Our first key observation is that the intersection property can be achieved as long as we have $\lceil\frac{n+t+1}{2}\rceil$ unique signatures. If we obtain this number of signatures out of $n = 2t+1$, safety is preserved in the sense that conflicting certificates cannot be formed by a malicious adversary. Of course, there are runs in which we cannot reach that threshold since $\lceil\frac{n+t+1}{2}\rceil > n-t$ (e.g., if t processes crash immediately as the run begins). But in this case, $f \geq \frac{t}{2}$, and $O(f)$ becomes asymptotically $O(t)$. Hence, we can use a fallback algorithm with $O(nt)$ communication complexity.

As we assume that $t \in \Theta(n)$, we can use Momose and Ren's synchronous algorithm that has $O(n^2)$ communication complexity [13] for the fallback. We denote that algorithm $\mathcal{A}_{fallback}$. Note that their algorithm is "stronger" than our proposed algorithm as it provides strong unanimity for validity (i.e., it solves strong BA). We can use their solution by checking the validity of $\mathcal{A}_{fallback}$'s output according to the predicate. If it is valid, this is the decision value, and otherwise a default valid value is decided. Equipped with these insights, we next present our algorithm.

During the phases part of the protocol, a correct process must commit a value before reaching a decision. When it has certainty about a value it updates that value in a *commit* variable, along a *commit_proof* of this commitment (a quorum certificate, signed by sufficiently many processes) and a *commit_level* indicating the latest phase of a valid commitment it heard of. Once a correct process commits to a certain value it can only commit to a value for which it heard a valid commitment proof in a later phase during the run. That is, it may decide on a value for which it did not send a commit message. Moreover, it may even decide on a value it did not commit at all. For example, if it reaches the fallback and no correct process has decided. Once a correct process reaches a decision it updates it in its local *decision* variable as well as a matching quorum certificate in *decide_proof* variable.

A single phase The code for a single phase is given in Algorithm 4. Each process p_i starts a phase with its initial value v_i and information about possible previous commits (*commit, commit_proof, commit_level*) and decisions (*decision, decide_proof*). Correct processes return with updated information about commits and decisions that were made in that phase (or prior to that). The guarantees of the phases are: (1) Every *decision* updated during a phase is valid; (2) All decisions updated by correct processes are the same and there exists at most one valid *decide_proof* in the system; and (3) If the phase's leader is correct, the phase is non-silent, and $n-f > \lceil\frac{n+t+1}{2}\rceil$, then all correct processes return with the same valid decision.

Every non-silent phase starts with the leader broadcasting a propose message with its value in line 32. Upon receiving this message, correct processes either vote for this value by signing it (line 34) or answer with a value that was previously committed as well as its commit quorum certificate and level (line 36). If the leader receives a committed value it

Algorithm 3 weak BA algorithm: code for process p_i with initial value v_i

```
    Initially $decision = undecided, bu\_decision = v_i, fallback\_start \leftarrow \infty$
    $decide\_proof, commit, commit\_proof, bu\_proof, fallback\_val, phase\_decision = \bot$
    $commit\_level \leftarrow 0$

 1: for $j = 1$ to $t + 1$ do
 2:     $phase\_decision, decide\_proof, commit, commit\_proof, commit\_level \leftarrow$
        $invokePhase(j, v_i, decision, commit, commit\_proof, commit\_level)$
 3:     if $decision = undecided$ and $phase\_decision \neq undecided$ then
 4:         $decision \leftarrow phase\_decision$
    Round 1:
 5: if $decision = undecided$ then
 6:     broadcast $\langle \mathsf{help\_req} \rangle_{p_i}$
    Round 2:
 7: if received $\langle \mathsf{help\_req} \rangle_{p'}$ message and $decision \neq undecided$ then
 8:     send $\langle \mathsf{help}, decision, decide\_proof \rangle_{p_i}$ to $p'$
 9: if received $t + 1$ messages of $\langle \mathsf{help\_req} \rangle_{p'}$ from different processes then
10:     batch these messages into $QC_{\mathsf{fallback}}(v)$ using a $(t + 1, n)$-threshold signature scheme
11:     broadcast the message $\langle \mathsf{fallback}, QC_{\mathsf{fallback}}, decision, proof \rangle_{p_i}$
12:     $fallback\_start \leftarrow now + 2\delta$
    Round 3:
13: if received $\langle \mathsf{help}, v, decide\_proof \rangle_{p'}$ with valid $v$ and $decide\_proof$ for $v$ and $decision =$
    $undecided$ then
14:     $decision \leftarrow v$
15: $bu\_decision \leftarrow decision$
16: while $fallback\_start > now$ do
17:     if received valid $\langle \mathsf{fallback}, QC_{\mathsf{fallback}}, v, proof_{p'} \rangle_{p'}$ then
18:         if $decision = undecided$ and $proof_{p'} \neq \bot$ is a valid proof for a valid $v$ then
19:             $bu\_decision \leftarrow v$
20:             $bu\_proof \leftarrow proof_{p'}$
21:         if $fallback\_start = \infty$ then
22:             broadcast the message $\langle \mathsf{fallback}, QC_{\mathsf{fallback}}, bu\_decision, bu\_proof \rangle_{p_i}$
23:             $fallback\_start \leftarrow now + 2\delta$
24: $fallback\_val \leftarrow \mathcal{A}_{fallback}$ with $\delta' = 2\delta$ and initial value $bu\_decision$
25: if $decision = undecided$ then
26:     if $fallback\_val$ is valid then
27:         $decision \leftarrow fallback\_val$
28:     else
29:         $decision \leftarrow \bot$
```

Algorithm 4 *invokePhase*$(j, v_i, \mathit{decision}, \mathit{decide\_proof}, \mathit{commit}, \mathit{commit\_proof}, \mathit{commit\_level})$: code for process p_i

```
30: $leader \leftarrow p_{j \mod n}$
    Round 1:
31: if $leader = p_i$ and $decision = \bot$ then
32:     broadcast the message $\langle \mathsf{propose}, v_i, j\rangle_{leader}$
    Round 2:
33: if received $\langle \mathsf{propose}, v, j\rangle_{leader}$ with a valid $v$ for the first time and $commit = \bot$ then
34:     send $\langle \mathsf{vote}, v, j\rangle_{p_i}$ to $leader$
35: else if received $\langle \mathsf{propose}, v, j\rangle_{leader}$ and $commit \neq \bot$ then
36:     send $\langle \mathsf{commit}, commit, commit\_proof, commit\_level, j\rangle_{p_i}$ to $leader$
    Round 3:
37: if $leader = p_i$ then
38:     if received $\langle \mathsf{commit}, w, QC_{\mathsf{commit}}(w), level_{\mathsf{commit}}, j\rangle_{p'}$ then
39:         broadcast the message $\langle \mathsf{commit}, w, QC_{\mathsf{commit}}(w), level_{\mathsf{commit}}, j\rangle_{leader}$ according to
    the maximal $level_{\mathsf{commit}}$ received
40:     else if received $\lceil \frac{n+t+1}{2} \rceil$ messages of $\langle \mathsf{vote}, v, j\rangle_{p'}$ then
41:         batch these messages into $QC_{\mathsf{commit}}(v)$ using a $(\lceil \frac{n+t+1}{2} \rceil, n)$-threshold signature
    scheme
42:         broadcast the message $\langle \mathsf{commit}, v, QC_{\mathsf{commit}}(v), j, j\rangle_{leader}$
    Round 4:
43: if received $\langle \mathsf{commit}, v, QC_{\mathsf{commit}}(v), level_{\mathsf{commit}}, j\rangle_{leader}$ and $level_{\mathsf{commit}} \geq commit\_level$
    and $level_{\mathsf{commit}}$ is valid according to $QC_{\mathsf{commit}}(v)$ then
44:     send $\langle \mathsf{decide}, v, j\rangle_{p_i}$ to $leader$
45:     $commit \leftarrow v$
46:     $commit\_proof \leftarrow QC_{\mathsf{commit}}(v)$
47:     $commit\_level \leftarrow level_{\mathsf{commit}}$
    Round 5:
48: if $leader = p_i$ then
49:     if received $\lceil \frac{n+t+1}{2} \rceil$ messages of $\langle \mathsf{decide}, v, j\rangle_{p'}$ then
50:         batch these messages into $QC_{\mathsf{finalized}}(v)$ using a $(\lceil \frac{n+t+1}{2} \rceil, n)$-threshold signature
    scheme
51:         broadcast the message $\langle \mathsf{finalized}, v, QC_{\mathsf{finalized}}(v), j\rangle_{leader}$
52: if received $\langle \mathsf{finalized}, v, QC_{\mathsf{finalized}}(v), j\rangle_{leader}$ then
53:     $decision \leftarrow v$
54:     $decide\_proof \leftarrow QC_{\mathsf{finalized}}(v)$
55: return ($decision, decide\_proof, commit, commit\_proof, commit\_level$)
```

simply broadcasts it. Otherwise, if it manages to achieve the required $\lceil\frac{n+t+1}{2}\rceil$ threshold of signatures, it can form a quorum certificate committing its proposed value (line 40).

Note that at this point the committed value is not “safe enough” to be decided by correct processes. Byzantine leaders may cause correct processes to participate in forming a commit certificate for more than one value. As correct processes that have decided do not initiate phases, they might never communicate without going through Byzantine leaders. Thus, we need another level of certainty, in the form of the finalize certificate (to be stored in *decide_proof*). Using the commit levels, we maintain the invariant that if a correct process receives a valid finalize certificate, then no finalize certificate on another value can be formed.

Thus, after a correct process learns about a committed certificate with a level higher than the previous commit, it sends a matching decide message to the leader (line 44) and updates the commit information accordingly. If the leader receives the necessary threshold of decide messages, it forms a finalize quorum certificate. Every process that receives such a certificate can safely return the certificate's value as its decision.

Main algorithm The BA algorithm is given in Algorithm 3, using the phase algorithm as a building block. In our algorithm, all correct processes eventually decide by updating their *decision* variable. However, they do not halt. In our BA algorithm, we start by executing n phases with a rotating leader, ensuring that every correct process has a chance to reach a decision before executing the fallback algorithm. After the phases end there are several possibilities. First, if there are at most $\frac{n-t-1}{2}$ Byzantine processes, all correct processes must have decided. If there are more Byzantine processes, it may be the case that some correct processes decided and others did not. This could happen, for example, if a Byzantine leader causes the single correct leader to decide and not initiate its phase. By the phase guarantees, we know that all correct processes that decide by this point, decide the same valid value.

To address the case where not all correct processes decided, we have processes that have not decided ask for help from all other processes (line 6). If a correct process has decided and receives a help_req message, it answers with a help message including the decision value along with its proof at line 8. Note that in this round, the number of messages sent by correct processes is linear in the number of help requests. Specifically, if only Byzantine processes send help_req messages, the number of answers is $O(nf)$ and independent of t.

We note that if $t+1$ help requests are sent, then at least one of them is sent by a correct process that did not manage to form quorum certificates when it served as leader. Thus, in this case, $f \in \Theta(t)$, and we can execute the fallback algorithm. To make sure that all correct processes participate in the fallback algorithm, a fallback certificate with $t+1$ signature is formed.

We now encounter a new challenge. We must have all correct processes start a synchronous fallback algorithm at the same time. However, an adversary can form the fallback certificate and deal it to only some correct processes. This scenario can happen, for example, if less than $t+1$ help_req messages are sent, and the adversary adds t help_req signatures of its own. We thus require a correct process that receives a fallback certificate to broadcast it (line 22). This ensures that whenever one correct process runs the fallback algorithm, all of them do, but may still cause different correct processes to start the fallback at different times. Nevertheless, we know that the starting time difference is at most the δ it takes the message to arrive. We therefore run the fallback algorithm with $\delta' = 2\delta$, ensuring that all correct processes enter a fallback round before any of them exits from it.

Another subtle point is making sure that the fallback algorithm does not output a decision value that contradicts previous decisions made by correct processes. For that reason we add another 2δ safety window between getting notified about a fallback and initiating it. Correct processes that broadcast the fallback certificate attach their decision value and a proof (if exists). In the 2δ safety window, processes that learn about a decision value in the system adopt it as the initial value for the fallback algorithm (line 17). Recall that $\mathcal{A}_{fallback}$ is a strong BA protocol. If a correct process decides v prior to the fallback algorithm, all other correct processes learn about v during the safety window. Then, by strong unanimity, they all decide v.

Note that if the decision returned from $\mathcal{A}_{fallback}$ is not valid then it must be that strong unanimity preconditions are not satisfied (since correct processes always have valid inputs) and a default value is returned. Furthermore, whenever the strong unanimity precondition is not satisfied, it follows that not all correct processes propose the same value. As a result, there must exist more than one valid value in the run (the different correct proposals). And the $\perp$ default value is a valid weak BA output.

6.1 Correctness

We start by proving some lemmas about the phase's guarantees. First, we prove that if the *decision* is updated in a given phase, then its new value is valid.

▶ **Lemma 9.** *If a correct process updates* decision *during invokePhase, then v is a valid decision value.*

Proof. If a correct process updates its *decision* value at line 53 of *invokePhase* then it must have received a finalized certificate signed by $\lceil \frac{n+t+1}{2} \rceil$ processes. Hence, at least one correct process p' signed the decide message for v at line 44. By the code, p' signed the decide message for v if it received a commit certificate signed by $\lceil \frac{n+t+1}{2} \rceil$ processes. Hence, at least one correct process p'' signed the vote message for v at line 34. By the code, this is possible only if v is a valid value (line 33). ◀

Next, we prove that all correct processes that update their *decision* variable do so the same value. Moreover, at most one valid *decide_proof* can exist in the system. That is, a Byzantine process cannot devise a *decide_proof* that conflicts with any other *decide_proof* known by correct processes.

▶ **Lemma 10.** *All correct processes that update* decision *during invokePhase return the same* decision. *In addition, at most one* finalize *certificate can be formed in all phases.*

Proof. Assume that a correct process p_i sets its decision value to v in phase l and another correct process p_j sets its decision value to w in phase $k \geq l$.

If $k = l$, then p_i and p_j set their decision value in the same round and they both receive a finalize certificate signed by $\lceil \frac{n+t+1}{2} \rceil$ different processes. At least one correct process signed both certificates and since correct processes sign at most one finalize message per phase, $v = w$.

For the case where $k > l$: in phase l, p_i receives a finalize certificate signed by $\lceil \frac{n+t+1}{2} \rceil$ different processes. Thus, at least $\lceil \frac{n+t+1}{2} \rceil - t \geq \frac{n-t+1}{2}$ correct processes updated their commit to v in that phase, along with a matching commit proof and *commit_level* $= l$ (line 47). Since these processes are committed to v, they do not vote for any value proposed by a leader in the following phases. Thus at most $n - t - \frac{n-t+1}{2} = \frac{n-t-1}{2}$ correct processes can sign a conflicting proposed value in any phase greater than l. Since $\frac{n-t-1}{2} + t < \lceil \frac{n+t+1}{2} \rceil$, in any phase greater than l, no process can collect $\lceil \frac{n+t+1}{2} \rceil$ signatures on any value other than v. Because processes that updated *commit_level* $= l$ do not accept commitments on values with $d < l$ (line 43), at most $\frac{n-t-1}{2}$ can send a decide message on a value committed in phase $d < l$. Thus, at most $\frac{n-t-1}{2} + t < \lceil \frac{n+t+1}{2} \rceil$ decide messages for $w \neq v$ can be sent. Finally, no process can form and send a valid finalize certificate and decide upon any other value. Thus, $v = w$.

◀

We prove next that once a correct process is the leader of a non-silent phase, all correct processes return the same valid decision value by the end of that phase.

▶ **Lemma 11.** *If a correct leader invokes invokePhase in phase k and $f < \frac{n-t-1}{2}$, then all correct processes return the same valid* decision *by the end of the phase and this decision is a proposal of a correct process.*

Proof. The leader broadcasts its value v to all processes. If there is a correct process p for which $commit \neq \bot$, it sends the message $\langle \mathsf{commit}, w, proof, j \rangle_p$ to the leader. If the leader receives $\langle \mathsf{commit}, w, proof, j \rangle_{p'}$ (from any process), it broadcasts in round 3 a commit certificate for w. Otherwise, since $f < \frac{n-t-1}{2}$, leader receives $\lceil \frac{n+t+1}{2} \rceil$ messages voting for v and broadcasts a commit certificate for v. Then, all correct processes send the leader a finalize messages on v or w. Again, the leader receives $\lceil \frac{n+t+1}{2} \rceil$ messages finalizing v and broadcasts a finalize certificate for v. Correct processes receive this message and update their *decision* and *decide_proof* accordingly. Then, by the code they all return v.

◀

We now prove the correctness of the main BA algorithm. The following two lemmas prove that although some processes may start executing $\mathcal{A}_{fallback}$ at different times, they all successfully execute the fallback algorithm.

▶ **Lemma 12.** *If some correct process executes the fallback algorithm in Algorithm 3, all correct process do so and they all start at most δ time apart.*

Proof. Let p be the first correct process that executes the fallback algorithm at line 24 of Algorithm 3 at time t. This means that at time $t - 2\delta$, p broadcasts the fallback certificate to all other processes (line 22). By synchrony, this certificate is guaranteed to arrive at all correct processes by $t - \delta$, causing them to execute the fallback algorithm by $t + \delta$ if they have not done so earlier. ◀

▶ **Lemma 13.** *Consider a synchronous algorithm $\mathcal{A}$. Let σ be a synchronized run of $\mathcal{A}$ defined as follows. Let t be the time that the first correct process starts executing $\mathcal{A}$ in σ. All correct processes start executing $\mathcal{A}$ by $t + \delta$. The round duration is 2δ. In round r that begins (locally) in t_r, round r messages are processed if they are received in the time window $[t_r - \delta, t_r + 2\delta]$. Then σ is a correct run of $\mathcal{A}$.*

Proof. Consider a process p that starts round r at time t_r^p. Let p' be another correct process that starts round r at time $t_r^{p'}$, and sends a message to p in round r. By assumption, $t_r^{p'} = t_r^p + \epsilon$ where $-\delta \leq \epsilon \leq \delta$, and a message sent by p' at $t_r^{p'}$ arrives at time t_a where $t_r^{p'} \leq t_a \leq t_r^{p'} + \delta$. Note that round r ends at p at time $t_{r+1}^p = t_r^p + 2\delta$. Hence, $t_r^p - \delta \leq t_a \leq t_r^p + 2\delta$, as needed.

◀

Next, we the following lemma states that if a correct process manages to reach a decision prior to the fallback algorithm, then this is the only possible decision. Moreover, this decision value must be a valid one.

▶ **Lemma 14.** *If some correct process decides v before executing the fallback algorithm, then all correct processes decide v and v is valid.*

Proof. If there exists a correct process p that decides at line 4, then by Lemma 10 and the code all processes that decide at line 4 decide v as well. Moreover, all other correct process that have not decided by line 5, send help_req messages. Process p answers them and they all decide at line 14. Otherwise, no correct process decides at line 4 and they all send help_req messages at line 6. Then, they all receive $t + 1$ help messages and by the code perform the fallback algorithm. In addition, by the lemma assumption, it must be that p decides v at line 14.

If correct processes execute the fallback algorithm, then by the code they all wait a time period of 2δ before the execution, during which they receive all decisions made by other correct processes and update *bu_decision* accordingly (line 24). Specifically, they receive v from p. It follows from Lemma 10 that *bu_decision* is updated with the same value at all correct processes. Thus, all correct processes execute $\mathcal{A}_{fallback}$ with the same input, and by strong unanimity they set *fallback_val* to v at line 24.

We now prove that v is valid. If p decides v at line 4, then it must have updated *decision* in the scope of the relevant phase. By Lemma 9 this value is valid. Otherwise, if p decides v at line 14, then the validity follows from the code. Hence, since v is valid, all correct processes decide it by line 27. ◀

Finally, we are ready to prove the required BA properties.

▶ **Lemma 15** (Agreement). *In Algorithm 3 all correct process decide on the same value.*

Proof. First, by Lemma 10, all correct processes that decide in line 4 decide the same value v. In addition, it follows from the same lemma that every correct process that decides at line 14 after receiving a valid finalize certificate decides v, as at most one finalize certificate can be formed.

It is left to show that if not all correct processes decide before the fallback algorithm at line 24, they still decide upon the same value. If at least one correct process p receives a fallback certificate it follows from Lemma 12 that all correct processes receive the certificate within at most δ time of p. Then, by the code, all correct process execute the fallback algorithm at line 24 and by Lemma 13 and the fallback algorithm solves strong BA, providing agreement. By Lemma 14, we get that processes that decide before running the fallback decide on the same value. ◀

▶ **Lemma 16** (Termination). *In Algorithm 3 all correct process decide.*

Proof. If not all correct processes decide before line 5 and no correct process receives a fallback certificate, it follows that less than $t + 1$ correct processes broadcast help messages at line 6. Hence, at least one correct process p has decided by line 5. Process p receives all of the correct help messages at line 7 and answers them at line 8. All correct processes that asked for help then decide at line 14.

It remains to examine the case that at least one correct process p receives a fallback certificate. It follows from Lemma 12 that all correct processes receive the certificate within at most δ time of p. Then, by the code, all correct process execute the fallback algorithm at line 24 and by Lemma 13 and the fallback algorithm solves BA, providing termination. ◀

▶ **Lemma 17** (Unique Validity). *In Algorithm 3 if a correct process decides v then either $v = \bot$ or $validate(v) = true$, and if $v = \bot$ then more than one valid value exists in the run.*

Proof. Let v be the decision value of a correct process in Algorithm 3. First, by lines 27 – 29 $validate(v) = true$ or $v = \bot$. We prove that if $v = \bot$, then at least two valid values exist in the run.

By the code, all processes execute the fallback algorithm with valid inputs (either their initial valid values, or a valid value they adopt at line 19). By strong unanimity of $\mathcal{A}_{fallback}$, if all correct processes start with the same valid value v', then v' must be the returned decision value. This contradicts the fact that $\bot$ is returned at line 29. Therefore, not all correct processes execute $\mathcal{A}_{fallback}$ with the same value. As they all execute the fallback algorithm with valid inputs, it follows that at least two valid values exist in the run. ◀

In addition, we need to prove that every correct process updates its *decision* at most once.

▶ **Lemma 18.** *In Algorithm 3 all correct processes decide at most once.*

Proof. Any correct process updates *decision* at line 4, line 14 or lines 27 – 29. In all cases, it only does so if *decision* = *undecided*. Since by the code it does not update *decision* to the value *undecided*, it follows that *decision* is updated at most once. ◀

From Lemmas 15, 16, 17, and 18 we conclude:

▶ **Theorem 19.** *Algorithm 3 solves weak BA.*

6.2 Complexity

We show that if $f < \frac{n-t-1}{2}$, correct processes never perform the fallback algorithm.

▶ **Lemma 20.** *If $f < \frac{n-t-1}{2}$, correct processes never perform the fallback algorithm.*

Proof. In Lemma 11 we prove that if a correct process is the leader of a non-silent phase and $f < \frac{n-t-1}{2}$, then all correct processes return the same valid decision. Since Algorithm 3 is composed of n phases, every correct process has a chance to invoke its phase and all correct processes decide by line 4. Assume by way of contradiction that there exists a correct process that invokes the fallback algorithm. By the code, it has received a fallback certificate. However, such a certificate can only be formed by $t + 1$ unique help_req signatures, meaning that at least one correct process sent a help_req message. But this is impossible if all correct processes decide by line 4. ◀

Each phase is composed of a constant number of all–to–leader and leader–to–all rounds. Thus, it incurs $O(n)$ words. Potentially, there are n phases. However, Lemma 11 proves that once a correct leader invokes $invokePhase()$ and the number of actual failures is $f < \frac{n-t-1}{2}$, all correct processes decide by the end of that phase. Since correct leaders that had already decided do not invoke their phases (their phases are silent), the number of invoked phases depends on f itself. Thus, all phases combined send $O(n(f + 1))$ words.

After n invokePhase invocations end, help request messages are sent only by correct processes that did not decide. By the above-mentioned lemma, it happens only if $f > \frac{n-t-1}{2}$.

In this case, $f = \Theta(n)$ and since $t = \Theta(n)$ it holds that $O(nf) = O(n^2)$. Correct processes that decide by this point answer directly to whoever sent them help requests, without affecting the asymptotic complexity. If some correct process receives a fallback certificate, another all-to-all round is added, keeping the complexity $O(n^2)$. All other communication costs are incurred in the fallback algorithm, whose complexity is also $O(n^2)$.

7 Strong BA: the failure-free case

Recall that the optimal resilience for strong BA is $n = 2t + 1$. In this section, we present a binary strong BA protocol that has a communication complexity of $O(n)$ in the failure-free case. Otherwise, it has complexity $O(n^2)$. The question of whether an adaptive protocol with $O(n(f + 1))$ complexity can be designed for strong BA with optimal resilience remains open.

In the algorithm, presented in Algorithm 5, a single leader first collects all initial values. Since we solve binary agreement, in the failure-free case there must be a value proposed by $t + 1$ different processes. Thus, the leader can use a threshold signature scheme to aggregate a quorum certificate on this proposed value.

As a second step, the leader sends this certificate to all processes and attempts to collect n different signatures on the value. If it succeeds, it broadcasts it. Every process that receives a signed-by-all certificate can safely decide upon its value. If a correct process does not decide, it broadcasts a fallback message. Every process that hears such a message, echoes it at most once, and executes $\mathcal{A}_{fallback}$ after 2δ time with 2δ-long rounds, as in Section 6.

7.1 Correctness

▶ **Lemma 21.** *If some correct process executes the fallback algorithm in Algorithm 5, all correct processes do so and they all start at at most δ time apart.*

Proof is similar to Lemma 12 in Section 6.

▶ **Lemma 22** (Agreement). *In Algorithm 5 all correct processes decide on the same value.*

Proof. First, as correct processes only sign one decide message, every process that receives $QC_{\mathsf{decide}}(v)$ receives the same quorum certificate. Thus, all correct processes that decide at line 14 decide the same v. If at least one correct process receives a fallback message then by Lemma 21, they all execute the fallback algorithm at most δ time apart. Thus, if at least one correct process decides at line 14, then all correct processes that have not yet decided learn about v in the 2δ safety window, and adopt it as their initial value for the fallback (line 23). It follows that all correct processes decide with the same input value v and by strong unanimity this is the only possible decision. ◀

▶ **Lemma 23** (Termination). *In Algorithm 5 all correct processes decide.*

Proof. If not all correct processes decide by line 14, then a correct process broadcasts a fallback message at line 17. It follows from Lemma 21 that all correct processes receive the certificate within at most δ time of p. Then, by the code, all correct processes execute the fallback algorithm at line 28 and by Lemma 13 and the fallback algorithm solves strong BA, providing termination. ◀

▶ **Lemma 24** (Validity). *In Algorithm 5 if all correct processes propose the same value v, then the output is v.*

Algorithm 5 strong BA algorithm: code for process p_i with initial value v_i

```
Initially $decision, proof, bu\_decision, bu\_proof, fallback\_val = \bot$
$fallback\_start \leftarrow \infty$

1: $leader \leftarrow p_1$
   **Round 1:**
2: send $\langle v_i \rangle_{p_i}$ to $leader$
   **Round 2:**
3: **if** $leader = p_i$ **then**
4:     **if** received $t + 1$ messages of $\langle v \rangle_{p'}$ for some $v$ **then**
5:         batch these messages into $QC_{\mathsf{propose}}(v)$ using a $(t+1, n)$-threshold signature scheme
6:         broadcast the message $\langle \mathsf{propose}, v, QC_{\mathsf{propose}}(v) \rangle_{leader}$
   **Round 3:**
7: **if** received valid $\langle \mathsf{propose}, v, QC_{\mathsf{propose}}(v) \rangle_{leader}$ **then**
8:     send $\langle \mathsf{decide}, v \rangle_{p_i}$ to $leader$
   **Round 4:**
9: **if** $leader = p_i$ **then**
10:    **if** received $n$ messages of $\langle \mathsf{decide}, v \rangle_{p'}$ **then**
11:        batch these messages into $QC_{\mathsf{decide}}(v)$ using a $(n, n)$-threshold signature scheme
12:        broadcast the message $\langle \mathsf{decide}, v, QC_{\mathsf{decide}}(v) \rangle_{leader}$
   **Round 5:**
13: **if** received valid $\langle \mathsf{decide}, v, QC_{\mathsf{decide}}(v) \rangle_{leader}$ and $decision = \bot$ **then**
14:     $decision \leftarrow v$
15:     $proof \leftarrow QC_{\mathsf{decide}}(v)$
16: **else**
17:     broadcast the message $\langle \mathsf{fallback}, \bot, \bot \rangle_{p_i}$
18:     $fallback\_start \leftarrow now + 2\delta$
19: $bu\_decision \leftarrow decision$
20: **while** $fallback\_start > now$ **do**
21:     **if** received $\langle \mathsf{fallback}, v, proof_{p'} \rangle_{p'}$ **then**
22:         **if** $decision = \bot$ and $proof_{p'} \neq \bot$ is a valid proof for a valid $v$ **then**
23:             $bu\_decision \leftarrow v$
24:             $bu\_proof \leftarrow proof_{p'}$
25:         **if** $fallback\_start = \infty$ **then**
26:             broadcast the message $\langle \mathsf{fallback}, bu\_decision, bu\_proof \rangle_{p_i}$
27:             $fallback\_start \leftarrow now + 2\delta$
28: $fallback\_val \leftarrow \mathcal{A}_{fallback}$ with $\delta' = 2\delta$ and initial value $bu\_decision$
29: **if** $decision = \bot$ **then**
30:     $decision \leftarrow fallback\_val$
```

Proof. Correct processes only send decide messages on values with valid propose quorum certificates. Note that such a quorum certificate can only be formed with $t+1$ unique signatures. Hence, if all correct processes propose the same value v, then the only possible propose quorum certificate is with v. As a result, the only possible decide quorum certificate is with v as well.

The fallback algorithm is executed with either the original initial values or with a value that has a corresponding decide quorum certificate. Thus, if correct processes execute the fallback algorithm, they all start with v and by strong unanimity of $\mathcal{A}_{fallback}$, the decision is v. ◀

Finally, we prove that every correct process updates its *decision* at most once.

▶ **Lemma 25.** *In Algorithm 5 all correct processes decide at most once.*

Proof. Any correct process updates *decision* either at line 14 or at line 30. In both cases, it only does so if $decision = \bot$. Since it does not update *decision* to the value $\bot$ at any step of the algorithm, it follows that *decision* is updated at most once. ◀

From Lemmas 22, 23, 24, and 25 we conclude:

▶ **Theorem 26.** *Algorithm 5 solves binary strong BA.*

7.2 Complexity

We show that if the run is failure-free, correct processes never perform the fallback algorithm.

▶ **Lemma 27.** *If $f = 0$, correct processes never perform the fallback algorithm.*

Proof. If all processes are correct then they all send their initial values to the leader at line 2. Since values are binary, and there are $n = 2t+1$ processes, there must be a value v such that the leader receives $t+1$ unique signatures on v. Then, the leader broadcasts a propose certificate on v (line 6). Every correct process that receives this certificate replies with a signed decide message at line 8. Since all processes are correct, the leader then receives n signatures and then broadcasts a decide certificate on v (line 12). All processes then receive this certificate and decide v at line 14. None of them sends a fallback message. ◀

By Lemma 27, if all processes are correct then they never perform the fallback algorithm, and there are 4 all-to-leader and leader-to-all rounds, with a total of $O(n)$ words. Otherwise, the complexity is the complexity of the fallback algorithm, which is $O(n^2)$.

8 Conclusions and Future Directions

We have presented solutions for both Byzantine Broadcast and weak Byzantine Agreement with adaptive communication complexity of $O(n(f+1))$ and resilience $n = 2t+1$. To construct the weak BA algorithm, we utilized a threshold on the number of signatures such that on one hand, this number is sufficient to ensure a safe algorithm with adaptive communication in case there are not "many" Byzantine processes. On the other hand, failing to achieve this threshold indicates that there is a high number of failures, which allows the use of a quadratic fallback algorithm.

This weak BA algorithm is taken as a black box to construct our adaptive BB algorithm. Here, we carefully choose the predicate for the validity property, to allow us to reduce one problem to the other. Finally, for strong BA we propose a binary solution with optimal resilience. Our solution is linear in n in the practically common failure-free case, and quadratic in any other case. The question of whether a fully adaptive strong BA with optimal resilience exists or not remains open.

While $n = 2t + 1$ is optimal for strong BA, this is not the case for BB and weak BA, where any $t < n$ can be tolerated[2]. Thus, another possible future direction is improving the resilience of an adaptive BB or adaptive weak BA to support any $t < n$. Our weak BA algorithm relies on the current resilience to satisfy that if $f > n - \lceil \frac{n+t+1}{2} \rceil$ then f is linear in t. Note that this remains true for any resilience of $n = \alpha t + \beta$, for $\alpha > 1, \beta > 0$ without compromising the intersection property required for safety. Should a quadratic solution for weak BA be developed, it could be used to improve the total resilience of our adaptive algorithm (instead of Momose and Ren's algorithm [13]).

References

1 Ittai Abraham, Guy Golan-Gueta, and Dahlia Malkhi. Hot-stuff the linear, optimal-resilience, one-message bft devil. *CoRR*, abs/1803.05069, 2018.

2 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*, pages 337–346, 2019.

3 Eugene S Amdur, Samuel M Weber, and Vassos Hadzilacos. On the message complexity of binary byzantine agreement under crash failures. *Distributed Computing*, 5(4):175–186, 1992.

4 Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In *International conference on the theory and application of cryptology and information security*, pages 514–532. Springer, 2001.

5 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*, pages 524–541. Springer, 2001.

6 Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

7 Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: Sub-quadratic asynchronous byzantine agreement whp. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

8 Yvo Desmedt. Society and group oriented cryptography: A new concept. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 120–127. Springer, 1987.

9 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

10 Danny Dolev, Ruediger Reischuk, and H Raymond Strong. Early stopping in byzantine agreement. *Journal of the ACM (JACM)*, 37(4):720–741, 1990.

11 Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3132747.3132757`.

12 Vassos Hadzilacos and Joseph Y Halpern. Message-optimal protocols for byzantine agreement. *Mathematical systems theory*, 26(1):41–102, 1993.

13 Atsuki Momose and Ling Ren. Optimal communication complexity of authenticated byzantine agreement. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[2] For weak BA, this stems from the resilience for external validity.

14 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear byzantine smr. In *34th International Symposium on Distributed Computing (DISC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

15 Victor Shoup. Practical threshold signatures. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 207–220. Springer, 2000.

16 Alexander Spiegelman. In search for an optimal authenticated byzantine agreement. In *35th International Symposium on Distributed Computing (DISC 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

17 Zhuolun Xiang, Dahlia Malkhi, Kartik Nayak, and Ling Ren. Strengthened fault tolerance in byzantine fault tolerant replication. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 205–215. IEEE, 2021.

Chapter 3

Conclusion and Open Questions

3.1 Conclusion

In this thesis, we have delved into the realm of distributed systems, which are computer systems comprising multiple independent processes working together towards a common goal. These systems have gained immense popularity, with billions of users relying on them daily. The inherent advantage of distributed systems lies in their ability to offer enhanced scalability, fault tolerance, and reliability when compared to centralized systems.

Our exploration encompassed various models with changing timing assumptions and different communication methods. The connecting thread of our work is dealing with Byzantine users. Unlike crash failures, where processes cease to respond, Byzantine failures encompass arbitrary deviations from the protocol. In this area, we aimed to develop distributed services capable of operating effectively even under the most adverse user behaviors.

The utilization of distributed systems has witnessed a surge in recent decades, with blockchains emerging as a prominent use case. These decentralized and distributed digital ledgers record transactions across a network of computers. As these systems gain traction among a growing user base, ensuring properties like reliability and scalability encounters new challenges and Byzantine behavior is more likely to happen. To address these challenges, we have extensively studied diverse algorithms and primitives of distributed computing (Figure 3.1). By doing so, we have shed light on the design and implementation of distributed services that excel in providing reliability and scalability in the face of arbitrary faults. In conclusion, this thesis has contributed novel solutions and insights in the field of distributed systems, specifically in the areas of Byzantine Agreement (BA) algorithms, leader-rotation mechanisms, and asset transfer primitives, as we now describe.

In the first work presented in Chapter 2.1 we addressed the challenge of breaking the quadratic barrier of asynchronous Byzantine Agreement. Indeed, we have presented the first sub-quadratic asynchronous BA algorithm. To accomplish this feat, we devised

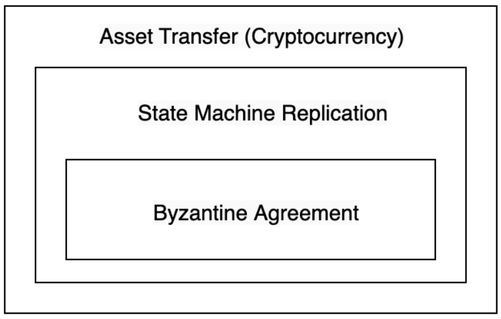

Figure 3.1: Relation between various services. Each box can be implemented using the primitives within it.

two key techniques that form the foundation of our algorithm.

The first technique involves the introduction of a shared coin algorithm, which relies on a trusted Public Key Infrastructure (PKI) and leverages Verifiable Random Functions (VRFs). This novel approach enables us to address the communication complexity hurdle associated with Byzantine Agreement in an asynchronous setting. By utilizing the shared coin algorithm, we can significantly reduce the overall computational costs involved.

Furthermore, we formalized the concept of VRF-based committee sampling specifically for the asynchronous model, marking a significant advancement in the field. This formalization provides a structured framework for sampling committees using VRFs, which is crucial for achieving sub-quadratic communication costs in our algorithm.

The algorithm we have developed only solves the Byzantine Agreement problem with high probability. It raises intriguing questions regarding the possibility of achieving a probability of 1 for certain properties while still maintaining the desired sub-quadratic communication cost. Exploring this possibility would contribute to a deeper understanding of the problem's inherent characteristics and potentially lead to further optimizations in future research.

The subsequent work that deals with different variations of the Byzantine Agreement problem (Chapter 2.4) focuses on the synchronous model and aims to redefine the possibilities under this model. In that work, we have presented novel solutions for both Byzantine Broadcast (BB) and weak Byzantine Agreement with adaptive communication complexity of $O(n(f+1))$ and resilience $n = 2t + 1$. In constructing the weak BA algorithm, we employed a threshold signature scheme, where the number of signatures is strategically chosen to serve two purposes. Firstly, this threshold ensures the algorithm's safety and adaptive communication complexity when the number of Byzantine processes is not significant. Secondly, failing to meet this threshold indicates a high number of failures, prompting the utilization of a quadratic fallback algorithm.

Taking advantage of the developed weak BA algorithm as a black box, we proceeded to construct our adaptive BB algorithm. The weak BA problem is defined, among other properties, by a validity predicate for the required decision output. When designing the BB protocol, careful consideration was given to selecting this predicate, allowing us to establish a reduction between the two problems. Furthermore, for a strong Byzantine Agreement, we proposed a binary solution with optimal resilience. In the absence of failures, our solution achieves linearity in n, which is practically desirable. However, in any other case, the communication complexity becomes quadratic.

In another work (Chapter 2.3) we shift and study the State Machine Replication problem. The SMR landscape has witnessed significant influence from blockchain advancements, with many cutting-edge blockchain-SMR solutions built upon two core pillars: Chaining and Leader-rotation that capture the changing leaders that drive the decisions within the protocol. However, the conventional round-robin mechanism employed for Leader-rotation presents an undesirable behavior, where crashed parties repeatedly assume the role of leaders, thereby impeding overall system performance.

In the discussed work, we have presented a new framework for Leader-Aware SMR that encompasses several desirable properties, notably formalizing the requirement of *Leader-utilization.* This requirement sets bounds on the number of rounds in crash-only executions where faulty leaders are allowed, thereby addressing the aforementioned limitation.

To achieve this Leader-Aware SMR, we have introduced Carousel, a novel reputation-based Leader-rotation solution. The key challenge in implementing adaptive Leader-rotation lies in the absence of consensus to determine a leader, as consensus itself depends on a designated leader. Leveraging the available on-chain information, Carousel determines a leader, ensuring system liveness despite this inherent difficulty. Through integration with a HotStuff implementation, Carousel has showcased remarkable performance enhancements. In faultless settings, it has achieved a throughput increase of over 2x, while in the presence of faults, it has provided an exceptional 20x increase in throughput and a 5x reduction in latency.

Finally, in Chapter 2.2 the wide interest in blockchains has led us to research the asset transfer primitive, which is the core problem solved by blockchains. We consider the shared memory model that is somewhat overlooked in this context. The exploration of the asset transfer problem in this model opened new research directions and expanded the understanding of concurrent shared-memory objects used by Byzantine processes. Our work led us to develop the concept of Byzantine linearizability, a specialized correctness condition designed to accommodate shared memory algorithms that can effectively withstand Byzantine behavior. Utilizing this notion, we undertook the task of establishing comprehensive upper and lower bounds for a variety of fundamental components in the field of distributed computing.

Through rigorous analysis, we demonstrated that atomic snapshot, reliable broadcast, and asset transfer are all problems that lack f-resilient emulations from registers when the number of processes n satisfies $n \leq 2f$. Conversely, we contributed an algorithm for Byzantine linearizable reliable broadcast, exhibiting a higher resilience threshold with $n > 2f$. Leveraging this algorithm, we further constructed a Byzantine snapshot with the same resilience capabilities. Notably, this Byzantine snapshot offers valuable applications, including the provision of a Byzantine linearizable asset transfer. Consequently, we established a tight bound on the resilience of emulations for asset transfer, snapshot, and reliable broadcast.

While our paper primarily focuses on feasibility results, we consciously chose not to delve into complexity measures. Specifically, our constructions assume unbounded storage. As a result, the matter of efficiency remains an open question, providing an opportunity for future research and exploration.

Overall, this thesis has significantly contributed to the field of distributed systems by introducing efficient BA algorithms, innovative leader-rotation mechanisms, and novel approaches to asset transfer in shared-memory models. The utilization of cryptographic tools, probability techniques, and careful analysis has provided valuable insights and practical solutions for building robust and reliable distributed systems. The findings presented in this thesis lay a solid foundation for future research and advancements in the field, empowering the development of more efficient and secure distributed systems.

3.2 Additional Open Questions

In the context of this thesis, where several research questions have already been presented, it becomes evident that additional avenues of investigation arise, encompassing both theoretical and practical aspects of distributed systems. These new directions hold the potential to further enrich our understanding and contribute to the advancement of the field. On the one hand, from a theoretical perspective, it is intriguing to explore questions that delve into the intricacies of mathematical models, seeking to refine existing frameworks, establish stronger theoretical foundations, and potentially identify novel complexity bounds. On the other hand, within the realm of practical system implementations, there is a pressing need to address challenges encountered in real-world scenarios, such as scalability, fault tolerance, and resource management. We present herein a collection of research directions that are worth exploring, in our opinion.

3.2.1 Towards Efficient Byzantine Agreement

Adaptive communication with optimal resilience. In our recent work [CKS23] we present an adaptive solution to Byzantine Broadcast and weak Byzantine Agreement with adaptive communication complexity of $O(n(f+1))$ and resilience $n = 2t+1$. While $n = 2t + 1$ is optimal for BA with strong unanimity, this is not the case for BB and weak BA, where any $t < n$ can be tolerated. Strong unanimity requires that if all correct processes propose the same value, this value must be decided. The challenge is that while the leader-based approach is commonly used to achieve low communication costs in synchronous protocols, it is unclear how to exploit it to solve strong unanimity. Delegating one process with a unique role (and consecutively its initial value) forbids us from acknowledging all initial values. On the other hand, requiring every process to spread its value has quadratic cost in the system before even trying to solve the actual problem.

In our work, we manage to solve BA with strong unanimity with resilience $O(n)$ in failure-free runs. This proves that there does not exist a lower quadratic lower bound on communication (as exists for the number of signatures), but the question of whether the adaptive solution can be designed for any number of failures $f < t$ remains open.

Multi-valued sub-quadratic Byzantine Agreement. In our first work [CKS20] we present the first sub-quadratic BA algorithm for an asynchronous message-passing environment. This result relies on a primitive called *shared coin*, that we implement in the paper, and returns all correct processes the same bit$\in \{0, 1\}$, with some constant probability. Due to the use of the coin, our BA solution only works for a binary domain. I.e., the decision value must be either 0 or 1. This has various uses in distributed systems, but in the context of blockchain systems, it cannot be used to decide upon blocks' order in the chain. To get one step closer to everyday use, we would like to solve a multi-valued sub-quadratic BA algorithm, where decisions can be from a larger domain. This is currently an open question.

Bibliography

[AGM18] Ittai Abraham, Guy Golan-Gueta, and Dahlia Malkhi. Hot-Stuff the Linear, Optimal-Resilience, One-Message BFT Devil. *CoRR*, abs/1803.05069, 2018.

[AMN$^+$20] Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Maofan Yin. Sync HotStuff: Simple and practical synchronous state machine replication. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 106–118. IEEE, 2020.

[AMS19] 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*, pages 337–346, 2019.

[BCC$^+$19] Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, François Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the Libra Blockchain. *The Libra Assn., Tech. Rep*, 2019.

[CKS05] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in Constantinople: practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18(3):219–246, 2005.

[CKS20] Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: sub-quadratic asynchronous byzantine agreement whp. In *34th International Symposium on Distributed Computing*, 2020.

[CKS23] Shir Cohen, Idit Keidar, and Alexander Spiegelman. Make every word count: adaptive byzantine agreement with fewer words. In *26th International Conference on Principles of Distributed Systems (OPODIS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

[CS20] Benjamin Y Chan and Elaine Shi. Streamlet: textbook streamlined blockchains. In *Proceedings of the 2nd ACM Conference on Advances in Financial Technologies*, pages 1–11, 2020.

[DR85] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *J. ACM*, 32(1):191–204, January 1985.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[FLP85] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[GHM$^+$17] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: scaling byzantine agreements for cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 51–68, 2017.

[GKL15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: analysis and applications. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 281–310. Springer, 2015.

[GLT$^+$20] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818, 2020.

[KS11] Valerie King and Jared Saia. Breaking the $O(n^2)$ bit barrier: scalable byzantine agreement with an adaptive adversary. *Journal of the ACM (JACM)*, 58(4):1–24, 2011.

[Kwo14] Jae Kwon. Tendermint: consensus without mining. *Draft v. 0.6, fall*, 1(11), 2014.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[MMR15] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with $t < n/3$, $O(n^2)$ messages, and $O(1)$ expected time. *Journal of the ACM (JACM)*, 62(4):31, 2015.

[MR20] Atsuki Momose and Ling Ren. Optimal communication complexity of byzantine consensus under honest majority. *arXiv preprint arXiv:2007.13175*, 2020.

[MRV99] Silvio Micali, Michael Rabin, and Salil Vadhan. Verifiable random functions. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 120–130. IEEE, 1999.

[Nak09] Satoshi Nakamoto. Bitcoin: a peer-to-peer electronic cash system, 2009.

[NBMS20] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: byzantine view synchronization. In *Proceedings of the Cryptoeconomic Systems Conference (CES'20)*, 2020.

[Rab83] Michael O Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409. IEEE, 1983.

[Spi21] Alexander Spiegelman. In search for an optimal authenticated byzantine agreement. In *35th International Symposium on Distributed Computing*, 2021.

[Tea] The Diem Team. Diembft v4: state machine replication in the diem blockchain. `https://developers.diem.com/docs/technical-papers/state-machine-replication-paper.html`.

[YMR+19] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

מטרת העבודה היא לפתח אלגוריתמים וטכניקות המאפשרים לספק את המאפיינים הרצויים של אמינות, עמידות בפני תקלות ומדרגיות במערכות מבוזרות שיכולות להיות מאומצות בעולם האמיתי. בפרט, השימוש הנרחב במערכות פותח צוהר לסוגים שונים של התנהגויות מצד משתמשים, אשר לכל משתמש תמריצים משלו. כך למשל, משתמש ברשת בלוקצ'יין עלול לנסות "לתחמן" את המערכת במטרה להרוויח יותר כסף לחשבונו האישי. התנהגויות מסוג זה נלכדות במודל הביזנטי, אשר מאפשר למשתמשים לפעול בצורה שרירותית. על כן, באמצעות בחינת מימוש של שירותים מבוזרים והתחשבות באתגרים הייחודיים שמציבות התקלות השרירותיות, אנו לוקחים חלק בהבנה והתקדמות החישוב המבוזר כולו.

התזה מתחילה בדיון על שלוש בעיות מרכזיות בחישוב מבוזר: שכפול מכונת מצבים, הסכמה ביזנטית ומטבעות קריפטוגרפים (Cryptocurrencies) . שכפול מכונת מצבים הוא שירות המורכב ממספר תהליכים הפועלים כמכונת מצבים יחידה ועמיד בפני תקלות. השירות מוודא שהתהליכים השונים עוקבים אחרי סדר זהה של מעברי המצב אשר נגרמים בעקבות בקשות של לקוחות. בעיית ההסכמה ביזנטית היא בעיה שבה קבוצה של תהליכים מנסים להגיע להחלטה משותפת, על אף הנוכחות של תהליכים ביזנטיים זדוניים. הבעיה האחרונה של מטבעות קריפטוגרפים מתארת את היכולת לנהל לקבוצת לקוחות את נכסיהם. הקשר בין הבעיות הוא כדלקמן: דרך אחת לפתור את בעיית המטבעות הקריפטוגרפים היא באמצעות שכפול מכונת מצבים, כך שמצב המערכת בכל רגע נתון מתאר את יתרת החשבון של הלקוחות השונים. את בעיית שכפול מכונת מצבים ניתן לפתור באמצעות מופעים רבים של בעיית ההסכמה הביזנטית, כך שבכל שלב ההחלטה שמנסים להגיע אליה היא מצב המערכת הבא.

כיוון שבעיית ההסכמה היא אבן יסוד בסיסית בתחום החישוב המבוזר, התזה מתחילה בתיאור הרקע של הסכמה ביזנטית, סקירת מודלים שונים של הנחות סנכרון זמנים וגבולות תחתונים של סיבוכיות הבעיה בהם. כמו כן, הרקע מדגיש את ההתקדמויות האחרונות באלגוריתמים ביזנטיים תת-ריבועיים באמצעות כלים של שימוש באקראיות וכלים קריפטוגרפים, המאפשרים להשיג ביצועים משופרים במונחים של התקשורת הנדרשת לפתרון הבעיה.

התוצאות שמוצגות בתזה זו כוללות הצגת אלגוריתם להסכמה ביזנטית אסינכרוני תת-ריבועי וכן אלגוריתם סנכרוני לבעיה שסיבוכיות הפתרון שלו היא אדפטיבית למספר התקלות בפועל. כלומר, במערכת בה אין תקלות, סיבוכיות התקשורת היא לינארית במספר התהליכים. בנוסף, עבודה אחרת בתזה מציע מנגנון "החלפת מנהיגים" בבעיית שכפול מכונת המצבים הנקרא Carousel. מנגנון זה מבטיח תכונת הוגנות המאפשרת ביצוע בקשות של לקוחות תקינים (שאינם ביזנטיים) מחד, ומאידך בריצה ללא תקלות ביזנטיות משפר את ביצועי המערכת מבחינת הזמן שלוקח לעבד בקשות של לקוחות. לבסוף, בעיית המטבעות הקריפטוגרפים נבחנת באופן המנותק מבעיית שכפול מכונת המצבים. אנו חוקרים את הבעיה במודל תקשורת של זיכרון משותף. במודל זה, אנחנו חוקרים לראשונה אובייקטים מקביליים אשר משמשים במקביל לתהליכים תקינים גם תהליכים ביזנטיים. תוך הגדרת תנאי נכונות חדש של אובייקטים מסוג זה, אנו מדגימים מימושים של snapshot, שידור אמין של מידע ומטבע קריפטוגרפי ומראים בנוסף מהם התנאים ההכרחיים למימוש כזה.

לסיום, התזה תורמת לתחום המערכות המבוזרות באמצעות טיפול באתגרים מרכזיים ומספקת פתרונות לעמידות בתקלות שרירותיות, מדרגיות ואמינות תוך בחינת מודלים שונים של הנחות סינכרוניות, ותוך שימוש בכלים קריפטוגרפים והסתברותיים נרחבים.

תקציר

מערכת מבוזרת היא מערכת מחשב בה מספר תהליכים עצמאיים פועלים ביחד כמערכת אחת על מנת להשיג מטרה משותפת. התהליכים יכולים להימצא במיקומים גאוגרפיים שונים והם מתקשרים אחד עם השני כדי להעביר מידע רלוונטי להשגת המטרה. מערכות מסוג זה משמשות מיליארדי משתמשים באופן יומיומי. דוגמאות נפוצות למערכות אלו הן שירותי דואר אלקטרוני או שירותי ענן.

היתרונות המרכזיים של מערכת מבוזרת לעומת מערכת ריכוזית הוא הבטחת אמינות גבוהה יותר, תכונת מדרגיות וכן היכולת שלה להתמודד עם תקלות. כך למשל, במערכת מבוזרת העובדה שתהליך יחיד נופל אינו גורם למערכת כולה להיות מושבתת והיא יכולה להמשיך ולשרת משתמשי קצה.

תכונת המדרגיות (סקלביליות) היא מאפיין חשוב של מערכות מבוזרות. היא מייצגת את היכולת של מערכת להתמודד בצורה אלגנטית עם כמות הולכת וגדלה של עבודה, או את היכולת לגדול כדי לתת מענה לגידול בכמות העבודה. כלומר, ניתן להוסיף משאבים כדוגמת תהליכים נוספים למערכת במטרה לשפר את ביצועי המערכת ו/או לתמוך במספר גדול יותר של משתמשים.

עוד מאפיין חשוב הוא עמידות בפני תקלות. מאפיין זה מייצג את יכולת המערכת להמשיך לפעול בצורה תקנית, הן מבחינת נכונות התוצאות והן מבחינת היכולת להתקדם, למרות כשלים שונים של תהליכים במערכת. על כן, חשוב לדעת כיצד לייצג ולתאר תקלות שונות באופן תיאורטי בכדי להצליח לבנות אלגוריתמים המספקים את המאפיין הזה.

מודלים תיאורטיים שונים של מערכות מבוזרות מתארים דפוסים שונים של תקלות המתרחשות בתהליכים. חלקם עוסקים בתקלות הנקראות "תקלות קריסה", שבהן תהליכים במערכת עלולים להפסיק להגיב, ואילו חלקם עוסקים בתקלות "ביזנטיות". תקלות ביזנטיות מתארות תהליכים שעשויים להתנהג באופן שונה ממה שנדרש מהם על פי הפרוטוקול. למשל, תהליכים ביזנטיים עלולים להפסיק לשלוח הודעות, להתעלם מהן, או אף לשלוח הודעות עם תוכן שונה מהמצופה מהם. כל זאת במטרה לחבל בנכונות הפרוטוקול ולגרום לכשל של המערכת. בשני המקרים, נהוג לתאר את התקלות באמצעות הנחה של יריב שמגדיר את דפוס התקלות, ולבנות אלגוריתמים אשר יהיו עמידים בפני הדפוס הגרוע ביותר האפשרי.

בעבודת תזה זו, אנו מבקשים לבחון שירותים שמיושמים במערכות מבוזרות ונוטים לתקלות ביזנטיות. שימוש בשירותים מסוג זה התרחב בעשורים האחרונים, במיוחד לאור השימוש החדש של טכנולוגיות בלוקצ'יין שהן רשומות דיגיטלית לכתיבת עסקאות בין חשבונות. שכיוון שקהל המשתמשים מערכות אלו הולך וגדל, כך גדלים גם האתגרים בהבטחת מערכות נכונות ויעילות, שמתמודדות עם הביקוש הגובר.

תודות

ברצוני להביע את תודתי ליועצתי, עדית קידר, על ההנחיה, התמיכה האינסופית והתובנות שלא יסולאו בפז.

אני אסירת תודה לקבוצת המחקר המסורה ולחבריי על שיתופי פעולה פוריים ומהנים במהלך המחקר שלי. הייתם חלק מרכזי במסע הזה.

אני רוצה לחלוק את הערכתי למשפחתי על העידוד והמוטיבציה המתמשכים. האמונה שלכם ביכולות שלי דחפה אותי לשאוף למצוינות.

לסיום, והכי חשוב, אני רוצה להודות לבן זוגי, ברק ולביתנו רומי. התמיכה שלך לאורך הדוקטורט שלי היא שאיפשרה את זה. האמונה האיתנה שלך בי הייתה הכוח המניע שלי. תודה מיוחדת לבן לוויתי- הכלב הנאמן, זאוס, על כך ששהית לצידי במהלך שעות עבודה ארוכות.

הכרת תודה מסורה לטכניון, מלגת מכון האסו פלאטנר, ולתכנית מלגות אדמס של האקדמיה הלאומית הישראלית למדעים על מימון מחקר זה.

המחקר בוצע בהנחייתה של פרופסור עדית קידר, בפקולטה למדעי המחשב.

מחבר/ת חיבור זה מצהיר/ה כי המחקר, כולל איסוף הנתונים, עיבודם והצגתם, התייחסות והשוואה למחקרים קודמים וכו', נעשה כולו בצורה ישרה, כמצופה ממחקר מדעי המבוצע לפי אמות המידה האתיות של העולם האקדמי. כמו כן, הדיווח על המחקר ותוצאותיו בחיבור זה נעשה בצורה ישרה ומלאה, לפי אותן אמות מידה.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחבר ושותפיו למחקר בכנסים ובכתבי-עת במהלך תקופת מחקר הדוקטורט של המחבר, אשר גרסאותיהם העדכניות ביותר הינן:

Keren Censor-Hillel, Shir Cohen, Ran Gelles, and Gal Sela. Distributed computations in fully-defective networks. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 141–150. ACM, 2022.

Shir Cohen, Idit Keidar, and Alexander Spiegelman. Not a coincidence: sub-quadratic asynchronous byzantine agreement WHP. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, 25:1–25:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

Shir Cohen and Idit Keidar. Tame the wild with byzantine linearizability: reliable broadcast, snapshots, and asset transfer. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, 18:1–18:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

Shir Cohen, Idit Keidar, and Oded Naor. Byzantine agreement with less communication: recent advances. *SIGACT News*, 52(1):71–80, 2021.

Guy Goren, Lefteris Kokoris-Kogias, Alberto Sonnino, Shir Cohen, and Alexander Spiegelman. Proof of availability & retrieval in a modular blockchain architecture. In *Financial Cryptography and Data Security - 27th International Conference, 2023*, 2023.

Shir Cohen, Rati Gelashvili, Eleftherios Kokoris-Kogias, Zekun Li, Dahlia Malkhi, Alberto Sonnino, and Alexander Spiegelman. Be aware of your leaders. In Ittay Eyal and Juan A. Garay, editors, *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*, volume 13411 of *Lecture Notes in Computer Science*, pages 279–295. Springer, 2022.

Shir Cohen, Idit Keidar, and Alexander Spiegelman. Make every word count: adaptive byzantine agreement with fewer words. In *26th International Conference on Principles of Distributed Systems (OPODIS 2022).* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2023.

Konstantinos Chalkias, Shir Cohen, Kevin Lewi, Fredric Moezinia, and Yolan Romailler. Hashwires: hyperefficient credential-based range proofs. *Privacy Enhancing Technologies Symposium (PETS 2021)*, 2021.

שירותים מבוזרים תחת מתקפה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

שיר כהן

הוגש לסנט הטכניון — מכון טכנולוגי לישראל
תמוז התשפ״ג חיפה יולי 2023

שירותים מבוזרים תחת מתקפה

שיר כהן