TÉCNICO LISBOA

# Using Randomized Byzantine Consensus To Improve Blockchain Resilience Under Attack

**Afonso Garcia Louro do Nascimento e Oliveira**

Thesis to obtain the Master of Science Degree in

**Information Systems and Computer Engineering**

Supervisors: Rodrigo Seromenho Miragaia Rodrigues
Henrique Lícias Senra Moniz

**Examination Committee**

Chairperson: José Luís Brinquete Borbinha
Supervisor: Rodrigo Seromenho Miragaia Rodrigues
Member of the Committee: Alysson Neves Bessani

**November 2021**

# Abstract

The rise in popularity of blockchains has lead to an increasing interest in the development Byzantine Fault-Tolerant (BFT) state machine replication systems for a variety of use cases and operational scenarios. A common approach when implementing these systems is to build on top of leader-based partially synchronous consensus protocols, which rely on critical assumptions about the underlying network in order to guarantee liveness. Alternatively, randomized protocols are able to avoid these pitfalls by operating over a fully asynchronous model. However, existing constructions still fall significantly behind, in terms of optimal performance, when compared to their deterministic counterparts. In this thesis we present Alea-BFT, an asynchronous BFT protocol, which leverages randomization in order to guarantee liveness without relying on timing assumptions about the underlying network and provides significant asymptotic and practical improvements over the state of the protocols in this model. We implemented Alea-BFT and compare its performance with HoneyBadgerBFT and Dumbo1/2, under a deployment scenario using up to 128 replicas, uniformly distributed across the globe. The experimental results demonstrate that Alea-BFT is able achieve multi-fold improvements over the remaining protocols, specially as the system size increases.

# Keywords

Consensus; Blockchain; Asynchronous; Randomization; Byzantine Fault Tolerance.

# Resumo

O aumento em popularidade das blockchains conduziu uma procura crescente no desenvolvimento de sistemas de máquinas de estado replicadas tolerantes a falhas Bizantinas (BFT) para uma variedade usos e cenários operacionais. Uma abordagem comum, na implementação destes sistemas, passa por construir sobre protocolos de consenso parcialmente síncronos com recurso a líderes, os quais requerem suposições críticas sobre o comportamento da rede de modo a garantir o seu correto funcionamento. Por outro lado, os protocolos randomizados são capazes de operar de um modo completamente assíncrono, no entanto as propostas existentes ainda ficam significativamente aquém, em termos de desempenho ótimo, quando comparadas com as alternativas determinísticas. Nesta tese, introduzimos o Alea-BFT, um protocolo BFT assíncrono, que utiliza não determinismo para garantir robustez independentemente das características da rede subjacente, e apresenta melhorias significativas tanto práticas como assimptóticas relativamente ao estado da arte para o modelo assíncrono. Finalmente, implementamos o Alea-BFT e comparamos seu desempenho com outros protocolos assíncrono (HoneyBadgerBFT e Dumbo1/2), num cenário real usando até 128 réplicas, uniformemente distribuídas geograficamente. Os resultados experimentais demonstram que o Alea-BFT é capaz de alcançar melhorias significativas em relação aos restantes protocolos, especialmente à medida que a escala do sistema aumenta, indicando maior escalabilidade.

# Palavras Chave

Consenso; Blockchain; Assíncrono; Randomização; Falhas Bizantinas.

# Contents

# List of Figures

# List of Tables



# List of Algorithms

# Listings

# Acronyms

| | |
|---|---|
| **TSS** | Threshold Signature Scheme |
| **ABA** | Asynchronous Binary Agreement |
| **MVBA** | Multi-valued Validated Byzantine Agreement |
| **RBC** | Reliable Broadcast |
| **CBC** | Consistent Broadcast |
| **PRBC** | Provable Reliable Broadcast |
| **VCBC** | Verifiable Consistent Broadcast |
| **ABC** | Atomic Broadcast |
| **ACS** | Asynchronous Common Subset |
| **TCP** | Transport Control Protocol |
| **WAN** | Wide Area Nework |
| **RTT** | Round Trip Time |
| **BFT** | Byzantine Fault-Tolerant |
| **SMR** | State Machine Replication |
| **FLP** | Fischer-Lynch-Paterson |
| **MAC** | Message Authentication Code |
| **HMAC** | Hash-Based Message Authentication Code |
| **FIFO** | First-In First-Out |
| **PRF** | Pseudorandom Function |
| **GDH** | Gap Diffie-Hellman |
| **PoW** | Proof-of-Work |
| **PoS** | Proof-of-Stake |

# 1

# Introduction

## Contents

## 1.1 Impetus

The rise in popularity of cryptocurrencies has led to an increasing interest in deploying blockchain based systems for a variety of use cases and operational scenarios. The need for higher performance guarantees and legal compliance forced organizations to steer away from the permissionless model of Bitcoin [3] in favor of permissioned solutions. In a permissioned scenario, where the identity of network participants is known and the adversary's power is strictly bounded by the fraction of nodes under its control, a classic Byzantine fault tolerant state-machine replication approach is often used as a building block to assemble blockchain systems.

The topic of Byzantine Fault-Tolerant (BFT) is not new [4], in fact it has been extensively studied over the last decades and a wide variety of solutions can be found in the literature. The traditional approach for BFT consensus algorithms follows two major design patterns: partial synchrony and leader-based. Assumptions regarding the underlying network allow partially synchronous protocols to escape the limitations of the Fischer-Lynch-Paterson (FLP) result [5] whereas relying on a leader to coordinate consensus instances allows for a decrease in the overall algorithm complexity. Protocols such as PBFT [6] and its derivatives mostly fit into the previous model and have been considered the standard in BFT consensus for years.

The problem with protocols in this family is that, by operating over a *partially synchronous* model, progress is only guaranteed during periods where these timing assumptions hold, causing performance to deteriorate or even completely stall during uncivil intervals [7]. Additionally a *leader-based* approach introduces a single point of failure into the system, for example a malicious leader can purposely slow down the system throughput up until the minimum threshold required to avoid being replaced [8, 9]. Finally, most development efforts focused mostly on optimizing performance under the assumption that failures do not occur [9], falling back to a more expensive strategies when forced to recover from failures.

We argue that, in spite of achieving impressive results under relatively controlled operational conditions, protocols in this class, underperform when subject to the more hostile deployment characteristics of blockchain systems, such as *wide-area networks* of mutually untrusting peers were nodes are not only allowed but actually expected to fail, in an attempt to slow down or subvert critical system properties, and advocate for a different approach based on non-determinism.

Consensus protocols based on randomization [10] bypass FLP by relaxing one of the defining properties of the consensus problem to be probabilistic, therefore being able to operate on a completely asynchronous model, eliminating the possibility for a malicious network scheduler to thwart performance. BFT solutions in this model have been around since 1983 [11, 12], but despite presenting characteristics that make them very interesting from a theoretical standpoint have usually been considered impractical due to high communication costs and expected termination time [13]. We believe that randomized BFT protocols are not only practical but in fact, due to their leaderless design and asynchronous network

model, present a more resilient solution than traditional deterministic, leader-base based approaches for deployment in adverse blockchain environments.

## 1.2 Contributions

The main focus of this thesis was to present a randomized BFT protocol capable of achieving better performance and scalability than existing solutions for the asynchronous model. Further bridging the gap between asynchronous and partially synchronous BFT protocols, while simultaneously providing higher resilience guarantees in the presence of adversarial network scheduling and malicious replicas. With these goals in mind, we designed and implemented a novel randomized atomic broadcast protocol called Alea-BFT, that presents the following characteristics:

- It provides optimal resilience for the Byzantine model, tolerating up to $f = \lfloor \frac{N-1}{3} \rfloor$ faulty processes out of $N$ total processes, where faulty processes are allowed to arbitrarily deviate from the protocol spec and even collude with each other in an attempt to subvert its properties.
- It is completely asynchronous, meaning that no assumptions are made regarding the delivery schedule of messages by the network. This property is crucial to ensure robustness under adversarial network conditions and prevent performance attacks based on timing assumptions.
- It sidesteps from a design approach based on an asynchronous common subset framework, where most of recent efforts in developing asynchronous BFT protocols have been concentrated, in favour of a novel architecture based on a two phased pipeline design.
- It provides significant asymptotic improvements over the state of the art protocols in this model. Particularly, both expected message and communication complexities can be reduced by a factor of up to $\mathcal{O}(N)$, while still terminating in constant time.

Our experimental evaluation of Alea-BFT concluded that it consistently outperforms existing asynchronous atomic broadcast protocols for any system scale, further bridging the performance gap between deterministic and randomized protocols, while still providing all the resilience guarantees characteristic of the asynchronous model.

## 1.3 Document Organization

This thesis is structured as follows: Chapter 2 contains an review of background concepts including blockchain, Byzantine consensus and randomization. Chapter 3 presents related work in the area of robust BFT protocols and non-deterministic consensus in the context of blockchain. Chapter 4 contains

a description for a novel asynchronous atomic broadcast protocol based on randomization. In Chapter 5 we present implementation details for our protocol. Finally, Chapter 6 contains our experimental evaluation and Chapter 7 summarizes our findings.

# 2

# Background

## Contents

This section contains an overview of some background concepts required for a better comprehension of the problem statement presented as well as the proposed solution.

## 2.1 Blockchain

A blockchain is a distributed data structure that maintains an append-only ledger of blocks, containing application specific operations, linked together by cryptographic primitives. The concept was initially introduced in Bitcoin [3], to address the double spending problem in the context of payments on a peer-to-peer network, and has since been applied to a variety of other use cases ranging from supply chain management to digital identification. Depending on the deployment scenario as well as requirements of the target application one must consider different blockchain design paradigms:

### 2.1.1 Permissionless, PoW-based Blockchains

Permissionless blockchains, also known as public, impose no participation restrictions allowing the network to operate in a completely decentralized and anonymous manner. This class of blockchain systems requires a Sybil [14] resistant consensus mechanism, in order to determine the next block to append to the ledger without interference from bad actors flooding the network. The Nakamoto Consensus [3] protocol addresses this issue by requiring peers to solve a cryptographic puzzle through a Proof-of-Work (PoW) algorithm in order to create a valid block and append it to the ledger. Since in this scenario the adversary's power is bounded by its total hash rate, an adversary is required to maintain control of more than half of the total computational power of the network in order to compromise the system.

### 2.1.2 Permissioned, Consensus-based Blockchains

On the other hand, permissioned blockchains, implement access control mechanisms to govern network participation. This approach is useful for securing a network composed by a set of participants that share a common goal but don't necessarily trust each other, such as large corporations or financial institutions. In this scenario the adversary's power is strictly bounded by the fraction of nodes under its control that are admitted to the system, allowing permissioned blockchains, such as Quorum [15] or Diem [16], to be build according to a traditional Byzantine fault-tolerant State Machine Replication (SMR) approach [17].

### 2.1.3 Permissionless, Hybrid-consensus Blockchains

Classic Byzantine consensus algorithms are unsuitable for the permissionless model for two main reasons. First they achieve consensus based on a threshold of node votes, therefore being vulnerable to

Sybil attacks on an environment where the identity of the participants is unknown. Secondly, these algorithms become less and less efficient as the number of nodes increases, being unable to support an unbounded number of participants. A recent line of research on hybrid consensus aims to attain, in the permissionless model, the performance guarantees of Byzantine consensus algorithms by combining them with a Sybil resistant mechanism such as PoW or Proof-of-Stake (PoS) responsible for controlling protocol membership [18].

## 2.2 State Machine Replication

A traditional approach for building fault tolerant distributed systems is based on State Machine Replication (SMR). In SMR, replicas are modeled as deterministic state machines, composed of a series of variables representing state, that transition between valid states via the atomic execution of commands. The basic intuition is that correct replicas, assuming a common initial configuration, evolve through different states in a consistent way by executing the same ordered sequence of commands. In order to achieve replica coordination a fault tolerant state machine system must ensure the following properties [19]:

- **Agreement:** Every correct state machine replica receives every request.
- **Order:** Every correct state machine replica processes received requests according to the same relative order.

The previous requirements can be fulfilled by a total order broadcast primitive, which has been proven to be equivalent to solving the consensus problem [20].

## 2.3 Consensus Problem

The consensus problem is a fundamental part of distributed systems research, and consists of getting a set of distributed processes to agree on a common value from a collection of initial proposals. A strong form of consensus must satisfy the following properties [21]:

- **Agreement:** All processes that decide do so on the same value.
- **Termination:** Every non-faulty process eventually decides.
- **Validity:** The decided value must have been proposed by some process.

The first and last properties are safety properties, ensuring that nothing bad happens, whereas termination is a liveness property stating that good things eventually happen [22]. Solving consensus in

the absence of faults is a trivial task. However, we are interested in exploring this problem in the presence of faulty processors, particularly in the Byzantine failure model where replicas may behave arbitrarily or even collude in an attempt to subvert the properties of the protocol. We explore this problem further in Section 2.5.

Solving consensus also provides a solution to a series of higher-level problems. An important one, in the context of SMR, is the problem of atomic broadcast which informally states that all correct processes must deliver the same sequence of messages. This has been proven to be solvable by running a series of consensus instances deciding on which message to deliver next making both problems equivalent. Atomic Broadcast (ABC) can be formally defined in terms of the the following properties [21]:

- **Validity:** If a correct process broadcasts a message $m$, then some correct process eventually delivers $m$.
- **Agreement:** If any correct process delivers a message $m$, then every correct process delivers $m$.
- **Integrity:** A message $m$ appears at most once in the delivery sequence of any correct process.
- **Total Order:** If two correct processes deliver two messages $m$ and $m'$, then both processes deliver $m$ and $m'$ in the same order.

In the paradigm of blockchain an atomic broadcast primitive can be used to establish a total ordering on appends to the ledger.

## 2.4 FLP Impossibility

The FLP impossibility result [5] states that there is no deterministic solution for the consensus problem, capable of simultaneously ensuring both safety and liveness properties, in an asynchronous model where nodes can fail silently. In order to circumvent this result researchers have devised extensions to the original system model where consensus is possible. The most popular extension approaches can be grouped into the following categories:

- **Timing assumptions:** Dolev, Dwork and Stockmeyer [23] showed that by adding timing assumptions, and therefore imposing restrictions on the order in which processes deliver messages, consensus becomes possible with up to $n$ faulty processes. The strongest synchrony assumption is the one of full synchrony, where there is a known upper bond on the delay of messages between correct processes. However, this model is usually considered too restrictive to be used in practice. A more realistic approach, that builds upon the assumption that systems are mostly asynchronous interleaved with periods of synchrony, was introduced by Dwork et. al [24] as the partially synchronous model. This model assumes the existence of either an unknown bound on message

delay or a known bound that only applies after some initial interval, known as global stabilization time.

- **Failure detectors:** Failure detection mechanisms allow processes to be notified about another process failure. By relying on this abstraction, even if unreliably, consensus becomes solvable since an elected coordinator is guaranteed to eventually respond. Chandra et. al [25] described the minimum guarantees that a failure detector must satisfy in order to bypass FLP.

- **Randomization:** Randomized models circumvent the FLP result by making one of the properties that defines consensus probabilistic. The most common approach is to extend a liveness property and run the algorithm through multiple rounds until its non deterministic nature becomes irrelevant. For example, by weakening the termination requirement to only be required with probability equal to one, the model is able to bypass FLP as non-terminating executions continue to exist, but only with a collective probability equal to zero. Another, more unusual, approach involves sacrificing a safety property instead.

The previous techniques are not mutually exclusive and can be used as either a complementary subsystem or combined into a hybrid approach. For example, Aguilera and Toueg proposed a protocol [26] that relies on a failure detector to solve consensus but can fallback to a randomized solution in case of oracle failure.

## 2.5 Byzantine Consensus

Byzantine Fault Tolerance (BFT) refers to the ability of a system to tolerate arbitrary behaviour from a subset of its participants without compromising its critical operational properties. Achieving consensus in the event of Byzantine failures was formally proposed by Lamport et al. as the Byzantine Generals Problem [4] and has been the target of extensive academic research ever since.

The FLP result is logically still valid in Byzantine model, forcing protocols to operate over the system models with stronger assumptions presented in Section 2.4. Earlier work on this topic considered a synchronous model with the first solution was presented by Pease et al. [27], and later improved by Dolev and Strong [28]. A resilience bound of $\lfloor \frac{N-1}{3} \rfloor$ was proven optimal for any Byzantine solution without digital signatures by Ben-Or [11].

As seen in Section 2.2, state-machine replication (SMR) relies on a consensus primitive to establish total order on client updates. In 1999 Castro and Liskov proposed PBFT [6], an efficient leader-based Byzantine SMR protocol for the partially synchronous model with $\mathcal{O}(N^2)$ message complexity and leader replacement sub-protocol. PBFT has since then been deployed in several systems and is

usually considered as the comparison baseline for protocols in this model. After PBFT, many other protocols appeared [8, 9, 29–32] each with a unique set of improvements over prior work, but mostly following two main design choices: a leader-based architecture, and requiring partial synchrony assumptions for liveness.

In leader-based consensus protocols, the system moves through a series of configurations known as views. Each view defines a replica (primary or leader) to be in charge of defining the order in which client requests should be executed by the system. A view change sub-protocol is responsible for nominating a new leader in order to prevent a Byzantine primary from thwarting system progress. This scheme allows for much more efficient communication patterns than their leaderless counterparts, where all replicas play the same role and individually try to reach a decision. However as we will see in Section 3.2, also opens the door for a series performance attacks exploiting the leader as a single point of failure.

As briefly mentioned in Section 2.4, timing assumptions are one of system model extensions that protocols can adopt in order to solve consensus despite the limitations of FLP. This sort of assumptions can come in different forms, with the two most common being the ones of partial and weak synchrony. Commonly, protocols manifest these assumptions as event triggering timeouts. For example, if replicas detect no progress has been made during the timeout period they trigger a view change operation evicting the current leader. The issue with protocols in this model, which we will further explore in Section 3.3, is that the liveness properties that define consensus can only be guaranteed during periods where these synchrony assumptions hold.

## 2.6 Randomization

As mentioned in Section 2.4, extending the system model using randomization can be used to escape the limitations of FLP. The most common approach is to extend the termination requirement to allow non-terminating executions with a collective probability of zero. The termination requirement can therefore be modified to reflect this probabilistic nature:

- **Probabilistic Termination:** Every non-faulty process eventually decides with probability one.

There are two ways to introduce non-determinism into the system. One is to assume the model itself is randomized and applicable operations on each state only occur probabilistically [33]. The other randomized algorithm approach considers a source of randomness located in the processes themselves. In this model, processes have access to coin-flip operations that return random binary values according to a certain probability distribution. All of the randomized protocols referenced in our work operate based on this second approach. Another important characteristic of these algorithms is that by operating over an asynchronous model they are completely decoupled from the concept of “real time”. For this

reason their running time is usually characterized based on the expected number of rounds required for termination [34].

In some of the first randomized protocols [11,35], processes were equipped with local coins, sources randomness isolated within the process, that would randomly return a binary value. Despite being easy to implement, as a simple random number generator would suffice, the values of local coins are independent between processes therefore requiring an exponential number of rounds before eventually converging. Rabin [12] showed that randomized agreement was solvable within a constant expected number of rounds with resource to a common coin, a distributed object capable of delivering a stream of random bits visible by all processes but unpredictable for an adversary. Rabin's implementation constructed this object based on Shamir's secret sharing algorithm [36], with secret shares being distributed by a trusted dealer, during a setup stage, and then combined during execution to produce random bits. A subsequent protocol ABBA [37], based on the common coin abstraction, was able to reduce the expected message complexity of randomized agreement to $\mathcal{O}(N^2)$ by employing threshold cryptography methods for the coin construction with $\mathcal{O}(1)$ expected time complexity. More recently Mostefaoui et al. [38] proposed a signature free randomized binary agreement agreement with optimal resilience and $\mathcal{O}(N^2)$ message complexity.

Despite raising a lot of interest from a theoretical standpoint, randomized algorithms have historically been overlooked for the implementation of practical applications due to their high expected time and message complexities [13]. For this reason their use case has mostly been restricted to solving binary consensus [37,39]. However this assumption that randomization is inefficient in practice disregards two important points. The first is that consensus protocols are usually not executed in isolation but instead in the context of a higher-level problem, such as atomic broadcast, that can provide faster termination guarantees (e.g. by having multiple processes propose the same value). Secondly, these algorithms have historically been evaluated under the assumption that a strong adversary has absolute control over the delivery schedule of messages in the network. This model, despite presenting theoretical value, is not very realistic as a real adversary usually does not possess this ability. More recent lines of research build upon these observations to implement protocol stacks that present efficient transformations from randomized binary consensus to higher-level problems such as multi-valued consensus, vector consensus and atomic broadcast [13,40].

# 3

# Related Work

## Contents

In this chapter we present existing work on the area of blockchain systems based on a classic Byzantine consensus algorithms and explore the pitfalls of these protocols when subject to performance degradation attacks as well as existing mechanisms proposed to minimize the impact of those attacks.

## 3.1 Blockchains with Byzantine Consensus

As previously mentioned in Section 2.1, traditional Byzantine fault tolerant consensus protocols can be used as a building block to assemble blockchain systems when the identity of participants is known, either as a standalone primitive or combined with a Sybil resistance mechanism in a hybrid consensus approach. Most modern permissioned blockchain systems (i.e., Hyperledger Fabric [41], Corda [42], Quorum [15], Tendermint Core [43], Diem [16]) rely on deterministic, leader-based consensus algorithms, such as HotStuff [44], in order to establish a global ordering on append operations to the ledger. As we will explore in Sections 3.2 to 3.3, protocols in this class present a series of characteristics that make them vulnerable to performance degradation under adversarial conditions, arguably making them unsuitable for a deployment in more hostile scenarios. In Chapter 4, we present a practical non-deterministic alternative that can be directly integrated in some of these blockchains that support plug and play consensus module, such as Hyperledger Fabric [41], for increased robustness.

## 3.2 Robust BFT

Traditional research in the field of BFT has been primarily focused on building fast BFT protocols, designed to maximize performance for the common case (i.e. in the absence of faults). Among these protocols, Castro and Liskov's PBFT [6] is usually considered as the performance baseline. Subsequent protocols built upon PBFT usually offer improvements by leveraging optimizations, such as optimistic execution of client requests, that despite providing excellent performance in the absence of faults impose significant recovery overheads [9].

Amir et al. [8] show that protocols in this class are vulnerable performance attacks, by presenting practical attacks on PBFT capable of slowing it down to unpractical levels. For example, a pre-prepare delay attack, which consists of a malicious leader purposely delaying the ordering of client requests without falling below the minimum threshold required to avoid being evicted. This attack exploits the fact that in PBFT, non-leader servers only place timeouts on the first request of their queue of pending requests therefore only requiring the primary to process a single request per timeout period, while delaying all the others.

These observations lead researchers to devise more robust BFT protocols capable of sustaining acceptable performance in the event of leader misbehaviour, without greatly compromising throughput

during civil periods. Next we present an overview of three robust BFT protocols: Prime [8], Aardvark [9] and Spinning [45].

### 3.2.1 Prime

The Prime [8] protocol introduces a pre-ordering phase with three communication steps, before a global-ordering phase based on PBFT, responsible for bounding the resources required by the leader to perform its functions. This is achieved by offloading most of the tasks required to establish an order on client operations to all of the servers as a group, allowing non-primary replicas to constantly monitor the performance of the primary expecting to receive ordering messages at a certain rate independently of the system load. The rate at which a primary should send order requests is unique to each replica and computed as a function of three parameters: (a) Round Trip Time (RTT) between the replica and the primary (measured periodically); (b) the time required to execute a batch of requests; and (c) a constant representing the variability of the network latency. If the primary fails to provide a flow of order messages at the expected rate then it is replaced.

An experiment by Aublin et al. [2] demonstrated that the robustness of Prime is heavily reliant on the correct monitoring of the current network conditions. They presented an attack in which a faulty primary, in collusion with a single malicious client, was able to degrade the system throughput down to $22\%$ of the values achieved in the fault-free case.

### 3.2.2 Aardvark

Aardvark [9] follows the same basic structure as PBFT but is enhanced with a series of mechanisms that provide increased robustness against Byzantine disruptions without heavily compromising performance during the common case. In order to prevent centralized control over the system's throughput by a potentially faulty primary, Aardvark performs view change operations regularly. Replicas continuously monitor the leader's throughput, voting for a view change if this metric falls below a specific threshold. The minimum threshold is initially set to $90\%$ of the maximum value observed during the last $N$ views and then, after a grace period of $5$ seconds periodically increased by a factor of $0.01$. This forces the current leader to constantly raise the bar on the throughput provided and eventually dictates its eviction restarting the process on a new view. If the system workload changes, the minimum required throughput adjusts itself over the next $N$ views to reflect this change. Additionally, replicas monitor the frequency of ordering messages sent by the primary against an heartbeat timeout. The protocol also employs a series of other robustness mechanisms such as resource isolation and signed client requests. Finally digital signatures are used to authenticate client requests, providing non-repudiation over said requests and therefore eliminating a number of expensive corner cases found in existing protocols that rely on

Message Authentication Code (MAC) authenticators.

Another experiment by Aublin et al. [2] aiming at evaluating the robustness of Aardvark under specially crafted attack scenarios concluded that as long as the system is saturated, the amount of damage a faulty primary can induce is limited, as the throughput expected by the replicas is close to the maximum system capacity. Under dynamic loads however, the performance degradation can be much higher. Consider a scenario where the system load is low, then the throughput value expected by the replicas is also low, as it only depends on the maximum value observed during the last $N$ views. If the system load increases suddenly, a malicious leader will be able to exploit the low expectations of the system and delay requests until the periodic increase in the required throughput causes the value to catch up with the load increase. In this scenario an attacker was able to achieve a maximum throughput degradation of $87\%$.

### 3.2.3 Spinning

Spinning [45] is another BFT protocol that similarly to Aardvark relies on frequent primary rotation as its main robustness mechanism, particularly the leader is replaced after ordering a single batch of requests. Its normal operation is similar to PBFT's, with three communication steps, however view change operations are no longer required since the views are always changing. Clients send requests to all replicas in the system, each replica starts a timer, with timeout value $\Delta$, and waits for an ordering message from the primary containing the request. If the timer expires before the ordering message is received the current primary is blacklisted (i.e., prevented from being elected as primary in the future), the value of $\Delta$ is doubled and another replica becomes the primary. Otherwise $\Delta$ is simply reset to its original value and the primary automatically replaced by the next non-blacklisted one. There is also an additional merge operation responsible for deciding whether or not to execute requests from views where timeouts expired. It is important to note that, contrary to what happens in Prime, the timeout value is not dynamically calculated during execution but instead statically defined as a system parameter.

Aublin et al. [2] proposed a practical attack on Spinning capable of degrading its performance to unacceptable levels. In this attack a malicious primary simply delays sending request ordering messages by $\Delta - \delta$, where $\delta$ is a very small value, drastically reducing the system throughput, while still avoiding being excluded from the pool of possible leaders. Practical experiments on the impacts of the previous attack showed that it was able to reduce the throughput of the protocol to as low as $1\%$ and $4.5\%$ of the values achieved in the fault free case, under the static and dynamic workloads, respectively.

**Table 3.1:** Performance degradation of robust BFT protocols under attack [2].

| **Protocol** | Prime | Aardvark | Spinning |
|---|---|---|---|
| **Degradation** | 78% | 87% | 99% |

As we can see by the analysis above and the results presented in Table 3.1 these robust BFT protocols, are not exactly as resilient to performance attacks as we would expect them to be. We argue that this is related to the fundamental aspect of the approach of trying to minimize the impact that a faulty primary might impose in performance instead of targeting the root of the problem, namely the leader-based architecture of these protocols. Additionally, in one way or another, all of the theoretical attacks presented involve manipulating the timing assumptions of the protocol, such as round trip times or timeout parameters. In Section 3.3 we will present a different class of robust BFT protocols that leverages randomization in an attempt to overcome the shortcomings of the previous approaches.

## 3.3 Asynchronous BFT

Almost all modern Byzantine fault tolerant systems, even those labeled robust, rely on a certain level of timing assumptions (such as partial or weak synchrony) in order to ensure liveness. To assess the possible impacts of a faulty network on the performance of BFT protocols based on partial synchrony (PBFT [6], Q/U [46], HQ [29] and Zyzzyva [30]), Singh et al. [7] conducted experiments over a series of simulated scenarios where replicas are correct but the network is faulty (e.g., latency and bandwidth variations, packet loss and misconfigured timers). The results achieved demonstrated that even in the absence of malice all previous protocols are sensitive to variations in network behaviour, with the throughput dropping to zero under certain conditions.

As seen in Section 2.6, randomization allows consensus protocols to operate over a fully asynchronous model, therefore eliminating the need for timing assumptions and the liveness issues associated with them. Additionally randomized protocols are naturally leaderless further improving resilience by eliminating the primary as a single point of failure. In order to prove the increases in robustness provided by randomized protocols, Miller et. al [47] devised an experiment where an adversarial scheduler with full control over the delivery of messages attempted to compromise the liveness properties of both PBFT and a novel randomized protocol HoneyBadgerBFT [47]. The experimental results showed that the scheduler indeed prevented PBFT from making any progress at all while HoneyBadgerBFT (and by extension any asynchronous protocol) was still able to continue executing operations.

Most recent attempts of implementing practical atomic broadcast protocols for the asynchronous model are instantiated based on an Asynchronous Common Subset (ACS) framework. In ACS every party proposes an input value, and outputs a common vector containing the inputs of at least $N - f$ distinct parties. Formally, an ACS protocol satisfies the following properties [48]:

- **Validity:** If a correct party outputs a set $V$, then $|V| \geq N - f$ and $V$ contains the inputs of at least $N - 2f$ correct parties.
- **Agreement:** If a correct party outputs a set $V$, then every correct party outputs $V$.

- **Totality:** If $N - f$ correct parties propose an input, then all correct parties produce an output.

Algorithm 3.1 illustrates the overall structure of how modern asynchronous atomic broadcast protocols use ACS, by having each replica propose a threshold encrypted batch of transactions into an ACS instance, and delivering a canonically sorted union of the transactions contained in the agreed-upon vector upon decrypting its contents (we defer a more detailed explanation of this reduction to the next section). Next we present an overview of existing state of the art asynchronous BFT protocols that follow this framework: HoneyBadgerBFT [47], BEAT [49], Epic [50] and Dumbo1/2 [1]:

### 3.3.1 HoneyBadgerBFT

HoneyBadgerBFT [47] is usually regarded as the first practical BFT protocol for the asynchronous model. The authors made the critical observation that atomic broadcast could be built based on an ACS framework by combining it with a threshold encryption scheme, following the structure of Algorithm 3.1. The protocol proceeds in epochs, every epoch each replica proposes a set of $\lfloor B/N \rfloor$ transactions, where $B$ corresponds a configurable batch size parameter, and delivers $\Omega(B)$ transactions. Proposals are encrypted, using the shared public key distributed during the trusted setup, before being passed as input to to an ACS instance. The output vector of ACS, consisting of at least $N - f$ encrypted proposals, is then subject to a threshold decryption round, where replicas share decryption shares for the proposals included in the vector, before being canonically sorted and committed. The use of threshold encryption prevents an adversary from selectively censoring transactions, by selecting which proposals to include in the ACS output vector, since replicas commit into delivering a certain set of proposals before the adversary learns about the particular contents of each one.

A particularly elegant aspect of HoneyBadgerBFT is its ACS construction, illustrated in Figure 3.1, resulting from the composition of two sub-protocols/phases Reliable Broadcast (RBC) and Asynchronous Binary Agreement (ABA). A reliable broadcast primitive allows a sender to disseminate a value across all replicas in the system, ensuring agreement on delivery in the sense that either all correct replicas deliver the same value or none deliver. Formally, RBC provides the following guarantees [21]:

- **Agreement:** If any two correct parties deliver $v$ and $v'$, then $v = v'$.
- **Totality:** If any correct party delivers $v$, then all correct parties deliver $v$.
- **Validity:** If a correct sender inputs $v$, then all correct parties deliver $v$.

Additionally, the binary agreement primitive used allows replicas to agree on the value of a single bit. We defer a more detailed explanation of ABA to Section 4.3.6.

During the broadcast phase, every replica starts an RBC instance in order to disseminate its proposal across all other replicas. In the agreement phase, $N$ parallel ABA instances are invoked in order to

**Algorithm 3.1** Reduction from ACS to ABC

**constants:**
  $N$ ▷ The total number of replicas.
  $B$ ▷ The batch size parameter.
  $PK$ ▷ The public key received by the dealer.
  $SK_i$ ▷ The private key for $p_i$.

**state variables:**
  $r \leftarrow 0$ ▷ The current protocol round.
  $buf \leftarrow \emptyset$ ▷ The FIFO queue of input transactions.

**procedure** START
  **while** true **do**
    // Step 1: Random selection and encryption
    $p \leftarrow buf[0 : \lfloor B/N \rfloor]$ ▷ Randomly select $\lfloor B/N \rfloor$ transactions.
    $v_i \leftarrow TPKE.Enc(PK, p)$

    // Step 2: Agreement on ciphertexts
    **input** $v_i$ to ACS $(r)$
    **wait until** ACS $(r)$ delivers $\{v_j\}_{j \in S}$, where $S \subset [1..N]$ **then**

    // Step 3: Decryption
    **for each** $j \in S$ **do**
      $e_{j,i} \leftarrow TPKE.DecShare(SK_i, v_j)$
      **multicast** $\langle \texttt{DEC}, r, j, i, e_j \rangle$
      **wait until** receive $f + 1$ messages in the form $\langle \texttt{DEC}, r, j, k, e_{j,k} \rangle$ **then**
        $y_j \leftarrow TPKE.Dec(PK, \{(k, e_{j,k})\})$

    // Step 4: Delivery
    $block_r \leftarrow sorted(\cup_{j \in S}\{y_j\})$ ▷ such that $block_r$ is sorted in a canonical order
    $buf \leftarrow buf - block_r$
    $r \leftarrow r + 1$
    **output** $block_r$

decide on a $N$-bit vector, where the $i$-th value indicates whether or not to include the proposal from replica $P_i$ in the final ACS output. A correct replica proposes $1$ into the $i$-th ABA instance upon RBC delivering the proposal from $p_i$, and abstains from proposing $0$ into any ABA until $N - f$ ABA have decided for one, ensuring the ACS output contains at least $N - f$ proposals. After terminating all $N$ ABA instances, a correct replica waits until it has RBC delivered proposals from all replicas whose corresponding ABA decided for $1$ before outputting a vector consisting of said proposals.

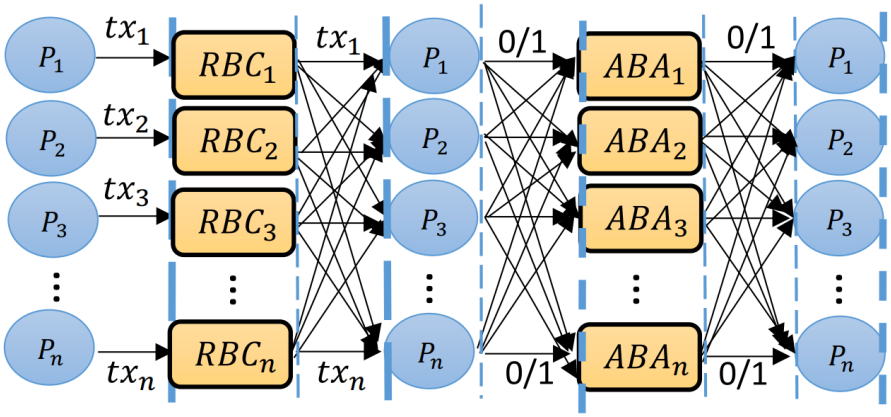



**Figure 3.1:** The structure of ACS in HoneyBadgerBFT [1].

HoneyBadgerBFT cherry-picks a bandwidth efficient RBC protocol using an erasure coding scheme (AVID broadcast) [51] and a state of the art signature free ABA [38] as the main building blocks of ACS. Note that the ABA protocol in the original HoneyBadgerBFT paper required a strong common coin that could not be realized from the threshold coin scheme used [37]. A revised version added a fix to this issue [52].

### 3.3.2 BEAT

BEAT [49] consists of a collection of five asynchronous BFT protocols (BEAT0-BEAT4) built on top of HoneyBadgerBFT. BEAT0, the baseline protocol of BEAT, presents a series of generic techniques aimed at optimizing HoneyBadgerBFT. Namely a more secure and efficient threshold encryption scheme, a direct implementation of threshold coin flipping and a more flexible and efficient erasure coding scheme. BEAT1 and BEAT2 introduce changes mostly focused on latency optimization. In BEAT1 the erasure-coded broadcast (AVID broadcast), used in HoneyBadgerBFT, is replaced with Bracha's broadcast [53], helping reduce latency for smaller batch sizes when the network contention is low. Additionally BEAT2 also moves the threshold encryption to the client side while still using Bracha's broadcast as in BEAT1. BEAT3 and BEAT4 implement BFT storage protocols where transactions are kept in the form of erasure coding, therefore despite presenting a significant performance improvement over the BEAT0-BEAT2

protocols cannot be used to general state machine replication.

An important remark is that most of the optimization techniques presented in BEAT are orthogonal and compatible with the other ACS framework based atomic broadcast protocols presented in this section, as the overall protocol structure is kept intact while simply replacing certain underlying building blocks with more suitable instantiations for certain deployment scenarios.

### 3.3.3 EPIC

EPIC [50] is an asynchronous BFT protocol built on top of BEAT0 that is secure for the adaptive corruption model. In this model the adversary can adaptively decide whose replicas to corrupt at any moment during the execution of the protocol, based on information accumulated so far (i.e., the messages observed and the states of previously corrupted replicas). On the other, hand previously presented protocols, such as HoneyBadgerBFT or BEAT, adopted a static corruption model, wherein the adversary must choose whom to corrupt at the start of the protocol. The reason for this comes from the fact that these protocols rely heavily on threshold cryptography schemes that despite efficient are not adaptively secure.

EPIC presents two major differences over BEAT0, making it adaptively secure: first, EPIC removes the need for threshold encryption via an hybrid transaction selection approach, where replicas select a random subset of transactions for most epochs but periodically switch to a First-In First-Out (FIFO) approach ensuring censorship resistance without relying on encryption. Secondly, it replaces the original Pseudorandom Function (PRF), used to instantiate a common coin, with a new, much more expensive, adaptively secure scheme [54] based on pairing-based cryptography.

Experimental results show that, despite still achieving reasonable performance, EPIC is less performant than its baseline protocol BEAT0 in terms of both latency and throughput, due to the more expensive cryptographic primitives employed in order to tolerate a corruption schedule more similar to traditional BFT protocols [6].

### 3.3.4 Dumbo1

Dumbo1 is the first protocol proposed in the recent work of Dumbo [1], a line of research aiming at reducing the number of `ABA` instances required to instantiate ACS, which has been identified by the authors as the main bottleneck of the construction presented in HoneyBadgerBFT [47]. This follows from the fact that, despite each ABA presenting a constant expected running time, the number of rounds required to execute $N$ concurrent instances grows to $\mathcal{O}(logN)$ [55], additionally these ABA instances are not fully concurrent as their starting time depends on on the delivery by distinct RBC instances. This presents a huge drawback specially as the system scale gets larger and the likelihood of very slow ABA

instances increases, which ultimately determine the running time of ACS in HoneyBadgerBFT.

To tackle this problem the authors of Dumbo1 present a redesigned ACS structure, illustrated in Figure 3.2, that only requires a small number $k$ of ABA instances, independent from $N$. The first phase of the ACS remains unchanged, with each replica relying on RBC to disseminate its proposal. The difference lies the agreement phase, where instead of investing an ABA instance per proposal, a small committee of $k$ aggregators is selected, such that at least one is honest with an overwhelming probability, to nominate which subset of proposals to output. After receiving the proposals from $N - f$, committee members broadcast an index set containing the ids of the received proposals through a second RBC instance. When a correct replica has received an index set $S_j$, from a committee member $P_j$, as well as all the corresponding proposals, it inputs $1$ into $\text{ABA}_j$. Replicas abstain from proposing $0$ into ABA until at least one of them has decided for $1$. After terminating all $k$ instances of ABA, a correct replica waits until it has received all index sets pertaining to committee members whose corresponding ABA instance for $1$ and finally waits for the proposals indicated by said index sets to be delivered, before merging them into the final output vector.

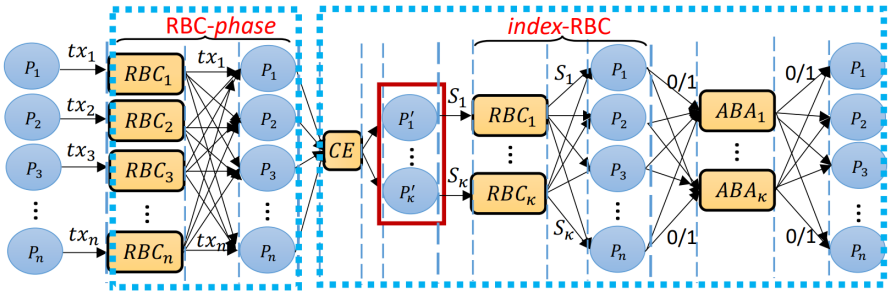



**Figure 3.2:** The structure of ACS in Dumbo1 [1].

At a first glance this ACS construction may appear to be slightly more expensive that the baseline instantiation of HoneyBadgerBFT, namely it introduces an extra RBC stage as well as a committee election sub-protocol. However, due the fact that in the second broadcast phase (index broadcast), each RBC is executed over a small index-set of size $N$ and the election of a $k$ committee members can be achieved via a distributed coin tossing protocol (a subroutine of ABA), these extra steps end up adding negligible overhead when compared with the benefit of reducing the number of ABA instances, particularly as the system scale becomes larger.

### 3.3.5 Dumbo2

Dumbo2 is the second protocol presented in Dumbo [1]. While in Dumbo1 the number of ABA instances required to instantiate ACS was reduced to a variable security parameter $k$, in Dumbo2 this value is further reduced into a constant. In order to achieve this, the authors explore the possibility of using Multi-

valued Validated Byzantine Agreement (MVBA) efficiently, instead of multiple concurrent ABA instances to construct the agreement phase of ACS.

An MVBA protocol allows agreement over an arbitrarily large set of values, that satisfy a validity predicate $Q$, instead of being restricted to a binary domain. Each party proposes a value, including some validation data, as input and outputs a common value that satisfies $Q$. Note that the output of MVBA does not need to have been proposed by a correct replica, as it's the case for ABA, only being constrained by $Q$. More formally, an MVBA protocol satisfies the following properties [1]:

- **Termination:** If every correct party inputs an externally valid value, then every correct party outputs a value.
- **External-Validity:** If a correct party outputs a value $v$, then $v$ satisfies the validity predicate $Q$.
- **Agreement:** If a correct party outputs a value, then every correct party that terminate output the same value.
- **Integrity:** Is a correct party outputs $v$, then $v$ was proposed by some party.

Instantiating ACS based on MVBA has traditionally been considered unpractical due to the high communication complexity of these protocols. For example the MVBA protocol in [56] has an expected communication complexity of $\mathcal{O}(N^2|m| + \lambda N^2 + N^3)$, where $|m|$ represents the MVBA input size, thus making $\mathcal{O}(N^2|m|)$ the dominating term for large input sizes and therefore unsuitable for directly constructing ACS. The the design of Dumbo2 is based on the observation that the above claim only holds when running MVBA over large input sizes. As a matter of fact, when $|m|$ is small, the overall communication complexity of MVBA does not exceed the one of the ACS construction in HoneyBadgerBFT, with the added benefit of only requiring a small constant number of ABA executions [56]. To leverage this property, RBC is replaced with an augmented protocol Provable Reliable Broadcast (PRBC) that further outputs a succinct size proof $\sigma$, such that whoever produces such proof can use it as guarantee that a correct replica has delivered the broadcast input. This allows for the broadcast phase of ACS to carry the bulk data while having the MVBA only executed over the constant size fingerprints generated during the previous phase.

As shown in Figure 3.3, the Dumbo2 instantiation of ACS is split in two stages. In the first stage, all replicas broadcast their proposals via PRBC and wait for $N - f$ instances to terminate. Let $W$ denote a vector, where each entry consists of a $(p_i, \sigma_i)$ tuple, such that an entry is put in $W$ if the corresponding PRBC terminates, and $Q$ the external validity predicate for $\texttt{MVBA}_r$. Predicate $Q$ ensures that $W$ contains at least $N - f$ distinct $i$ entries that satisfy $p_i \neq \perp$, and that each $(p_i, \sigma_i)$ is a valid output of the corresponding PRBC instance for that round. In the second stage replicas input their $W$ version into MVBA and wait for it to output a common $\overline{W}$ value, validated by $Q$. Finally correct replicas wait until

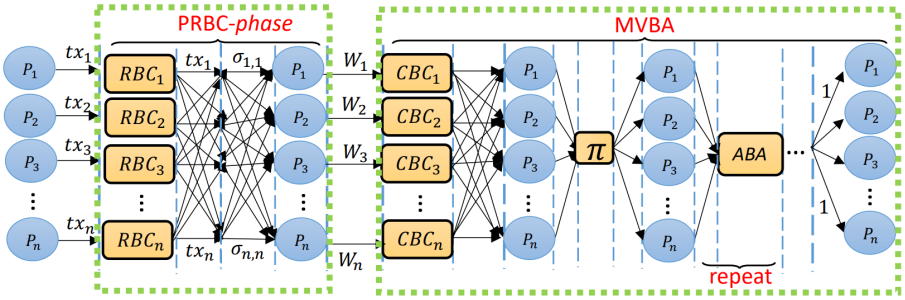



**Figure 3.3:** The structure of ACS in Dumbo2 [1].

they have delivered the input values for all PRBC instances contained in $\overline{W}$ and output these proposals for the ACS.

All protocols described above provide high robustness guarantees in the event of adversarial network conditions but their $\mathcal{O}(N^3)$ unamortized message complexity during normal operation poses a trade-off in terms of scalability (Table 6.1 summarizes their asymptotic complexities). Additionally they all fit into the same ACS framework, which despite allowing for a lightweight reduction into atomic broadcast still presents some limitations. In Chapter 4, we will explore these issues and present a proposal for a randomized BFT protocol that completely sidesteps from the ACS framework in an attempt to improve the message complexity by a factor of up to $\mathcal{O}(N)$, further bridging the gap between partially synchronous and asynchronous BFT protocols while providing a better balance in the trade-off between robustness and scalability.

# 4

# Protocol

## Contents

In this chapter we present Alea-BFT, a novel asynchronous atomic broadcast protocol for the Byzantine model. In Section 4.1 we present an overview of the protocol as well as the intuition for our design choices. Section 4.2 contains a description of the system model and Section 4.3 details the underlying primitives used as building blocks when instantiating the protocol. Section 4.4 describes the operation of the multiple components of Alea-BFT accompanied by the respective pseudocode. Section 4.5 presents an efficiency analysis, according to time, message and communication complexity metrics. Finally, in Section 4.6, we provide a formal security analysis of the protocol.

## 4.1 Overview

Recent efforts in implementing practical asynchronous Atomic Broadcast (ABC) protocols, have mostly concentrated efforts around an ACS framework, as detailed in Section 3.3. Protocols in this framework despite having been proven as practical [47], disproving the conception that asynchronous BFT protocols are naturally inefficient, and further refined in subsequent work [1, 49, 50], still present some significant drawbacks:

- **Message and communication complexities.** In ACS every replica must propose a candidate value out of which a subset is selected to be included in the final output vector. This requires all replicas to execute an all to all communication phase during which they disseminate their proposals. By itself, this step incurs a cubic message and communication costs due to the $N$ executions of RBC.

- **Number of ABA instances.** In order to operate asynchronously, the FLP impossibility result [5] requires the agreement protocol ABA to be randomized. Previous work [1] has shown that despite the expected number of rounds per ABA being constant, the overhead of running multiple ABA instances presents the major bottleneck for the protocols presented in Section 3.3. In this same work the authors were able to reduce the number of ABA executions required to instantiate ACS into a constant value at the cost of extra communication steps, but we argue that this value could be further reduced into a single execution without compromising the performance for smaller system scales.

- **Bandwidth usage.** As shown in Algorithm 3.1, atomic broadcast can be achieved by sequentially executing independent ACS instances. This means that some of the broadcast instances were effectively ”wasted” as their values were not included in the final output and therefore must be re-broadcast in a subsequent ACS execution, which results in a sub-optimal usage of system resources, particularly bandwidth.

- **Byzantine performance faults.** The adversary has absolute control over which replicas proposals are included in the final output, only constrained by the validity property of ACS. This means that up to $f$ proposals can originate from Byzantine replicas which can result in serious performance degradation, by for example submitting empty/invalid proposals.
- **Threshold encryption.** A reduction from ACS to ABC requires threshold encryption in order to ensure fairness. This adds an extra all to all communication step, during which replicas broadcast decryption shares for the proposals included in the ACS output vector, as well additional computation costs that could ideally be avoided.

As presented above, asynchronous protocols in the ACS framework present some significant drawbacks that impact their overall performance. However, it's understandable why most recent proposals follow this architecture. First these protocols are relatively simple, resulting from the composition of only a few basic primitives and with very little deployment specific parameters to tune. Second, they are completely leaderless, as every process proposes into every round, which provides very strong robustness guarantees. We argue that we can obtain significant performance improvements, by adopting a different protocol architecture from ACS without the leaderless operation model, while simultaneously providing similar robustness guarantees.

To explore this idea, we propose Alea-BFT, a novel ABC construction for the asynchronous model that completely sidesteps the ACS framework. The main idea when designing Alea-BFT is to have a single replica propose a value per consensus instance, similarly to what happens in leader based protocols for the partially synchronous model, while all others simply agree on whether to deliver it or not. Therefore, allowing us to remove the all to all communication phase with the added benefit of only requiring a single ABA execution per epoch.

A strawman proposal for our protocol could be adapted from ACS construction of HoneyBadgerBFT, presented in Section 3.3, but instead of having all replicas simultaneously propose candidate values, a single replica is selected as the proposer for each round. The role of the proposer is to select a value/batch proposal from its pending buffer and broadcast it to all replicas. Correct replicas would proceed to execute a single ABA to determine whether or not to deliver the proposed value for that round. Additionally, the proposer could be rotated upon every ABA execution, emulating the behaviour of fast leader rotation protocols [45], in order to address the scenario where the proposer is faulty.

There is however, one major issue with this strawman protocol, that makes in unsuitable for deployment in our system model. In an ACS framework, replicas are guaranteed to receive proposals from at least $N - f$ correct replicas, and therefore can wait until this threshold is achieved before deciding which values to input for the agreement stage that follows. Contrarily, in our strawman protocol only a single replica takes the role of the proposer at any given time, so there is no way to determine whether the current proposer is faulty or not, consequently making it impossible to effectively decide which value to

input into the ABA without resorting to some sort of timeout, which contradicts the asynchronous model.

The impossibility to wait for a specific threshold to be met before deciding on which value to input to the ABA, lead us to the idea of not waiting at all, by persisting undelivered proposals across rounds, such that every time a particular replica is reelected as the proposer, the corresponding ABA execution will decide over its backlog of pending proposals instead of a single newly proposed value. This way, replicas can propose into the ABA as soon as they enter a new round, as even in the case of a decision for 0, the same proposal will be eventually revisited when the same leader is elected and a larger threshold of replicas become aware of the proposal, guaranteeing a convergence for an ABA decision for 1 over time. In Alea-BFT, we leverage this idea to decompose the monolithic architecture of HoneyBadgerBFT-ACS, in which a binary agreement instance actively waits for the corresponding broadcast to terminate, into a two stage pipeline where the results of the first phase (broadcast component) are queued to be eventually processed by an execution of the second phase (agreement component).

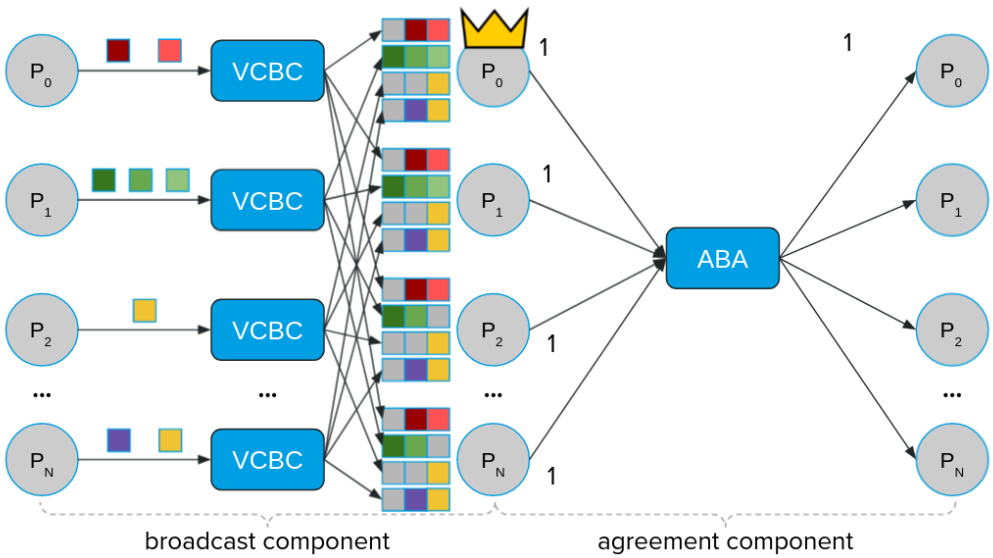



**Figure 4.1:** The protocol flow in Alea-BFT.

The resulting overall protocol flow is depicted in Figure 4.1. It starts with replicas receiving client requests and storing them in pending buffers of size $B$, when the buffer is full a replica disseminates its contents via a Verifiable Consistent Broadcast (VCBC) primitive, tagged with an incremental sequence number $s$, clears the buffer and becomes prepared to repeat the same process for $s+1$. This corresponds to the broadcast component of the Alea-BFT pipeline, in which all replicas constantly act as proposers in order to create locally ordered backlogs of undelivered proposals, replicated across all replicas, to be further processed by the next component of the pipeline. Note that every replica maintains $N$ backlogs of undelivered proposals, one for each replica in the system, which grow over time depending on how efficiently the agreement component is able to process them. The agreement component can be seen as

the coordinator between backlogs of pending proposals, constantly deciding which proposer's local order to follow next, by iteratively selecting one of these backlogs and deciding on whether or not to deliver the oldest proposal (by incremental sequence number) contained in it. To do so, replicas participate in a single ABA execution, voting $1$ if their backlog contains this proposal, or $0$ otherwise. If the ABA decides for $1$, this indicates that a sufficient threshold of correct replicas are aware of the proposal and may safely deliver it, as all other replicas are guaranteed to be able to actively fetch it if needed, via a recovery mechanism described in Section 4.4. Contrarily, in case of a decision for $0$ the agreement component simply selects a different backlog repeating the same process all over again.

## 4.2 System Model

We consider a network composed of $N$ processes, uniquely identified from the static set $S = \{P_0, ..., P_{N-1}\}$ out of which up to $f$ may fail, as well as an unbounded number of clients. Our protocol provides an atomic broadcast channel characterized by local `SEND` and `DELIVER` events parameterized by a payload value. A replica may execute `SEND` an arbitrary number of times, triggered in response to client requests, and must be prepared to `DELIVER` as many messages as the atomic broadcast channel outputs.

- **Byzantine faults.** We assume a Byzantine failure model where up to $f = \lfloor \frac{N-1}{3} \rfloor$ processes, an optimal threshold for this fault model, are allowed to fail during the execution of the protocol. We refer to these processes as faulty or corrupt. The adversary, a malicious entity whose goal is to subvert the protocol's properties, is given full control over the behaviour of these corrupt processes meaning that they can stop, deviate arbitrarily from the protocol's specification and even collude among each other in order to subvert the properties of the protocol. The remaining processes that do not fail during protocol execution are labeled as honest or correct.

- **Asynchronous network.** The network is asynchronous, meaning the delivery schedule of messages is delegated under adversarial control without bounds on communication delays or processing times. We consider the processes to be fully-connected by reliable channels, providing sender authentication and guarantees that messages are eventually delivered and no modifications to the messages occur in the channel.

- **Static corruptions.** As previously mentioned we assume that there are up to $f$ faulty nodes under adversarial control. In a static corruption model the adversary must determine which replicas to corrupt before the start of the protocol and cannot change this set latter on. It gets complete access to all the faulty nodes initial internal states and can also make them arbitrarily misbehave during the execution of the protocol.

- **Computational model.** The computational power of the adversary is said to be bounded, meaning it is constrained to perform polynomial-time operations and therefore unable to subvert the cryptographic primitives employed.

- **Trusted Dealer.** A trusted third party (dealer) is required to bootstrap the initial state of all replicas. Particularly, to generate and distribute the secret shares and verification keys for the threshold signature scheme used. Dependence on a trusted dealer only holds during a protocol setup phase, after which an unlimited number of messages can be processed.

## 4.3 Building Blocks

In this section we present some baseline definitions and describe the underlying primitives that are used as building blocks when implementing the Alea-BFT protocol. Additionally, we also provide specific algorithms for instantiating each one of the primitives.

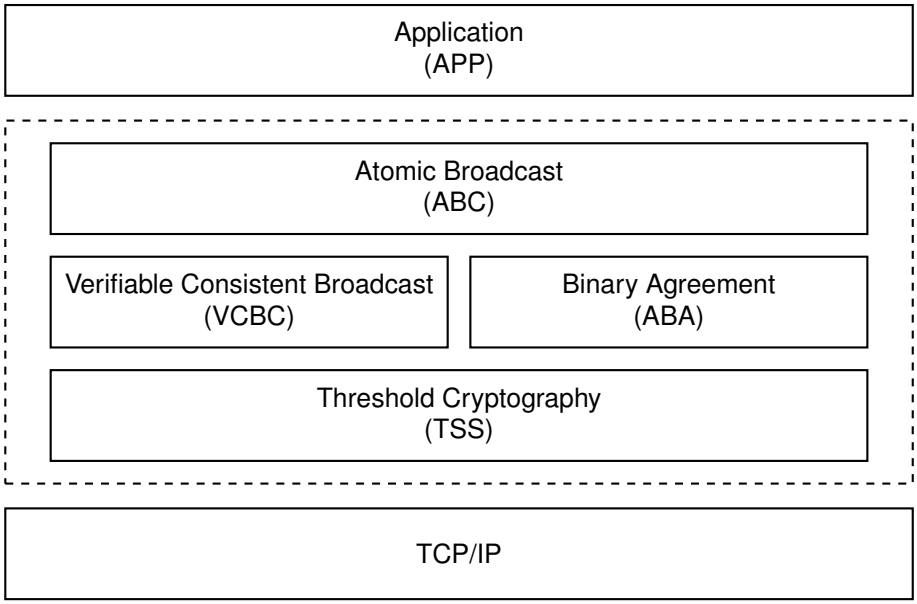



**Figure 4.2:** Building blocks of Alea-BFT.

As shown in Figure 4.2 and similarly to other protocols in this setting we provide an higher level protocol for atomic broadcast which invokes sub-protocols to carry out certain tasks. In this modular stack architecture, upper level protocols can provide input and receive output from sub-protocols down the stack. Additionally, since multiple (sub) protocol instances can be executed simultaneously, messages pertaining to a particular instance are isolated by tagging each one with an unique protocol identifier which can latter be used for routing inside the protocol stack. We use parenthesis to distinguish

between tagged instances of each sub-protocol, which can also piggyback information relevant to the protocol execution. For example VCBC $(p, s)$ corresponds to a verifiable consistent broadcast identified by the pair $\langle p, s \rangle$, corresponding in this case to the sender $p$ and the priority value $s$ attributed by process $p$ to the value being broadcast. For succinctness, we omit these ID tags from our messages whenever possible.

### 4.3.1 Threshold Cryptography.

Threshold cryptography is crucial for several of the sub-protocols in Alea-BFT and forms a core component of its architecture. Particularly, Alea-BFT relies threshold schemes for digital signatures and coin-tossing. They are all non-interactive and therefore do not require any particular communication pattern, being easily integrated into an asynchronous protocol.

### 4.3.2 Threshold Signature Scheme

A Threshold Signature Scheme (TSS) is a cryptographic primitive that allows for multiparty key generation and signing. A $(t, n)$-threshold signature scheme allows a subset of $t$ out of $n$ participants to generate a valid signature while disallowing its creation when the number of protocol participants is smaller than $t$. In involves distributing shares of a signing key $sk_i$ to each of the $N$ parties as well as a common public key $mpk$ and a public key vector $PK$. A $(t, n)$-TSS must provide two basic security requirements:

- **Unforgeability**: It is infeasible for a polynomial-time adversary to output a valid signature on a message that was submitted as a signing request to less than $n - t$ honest parties.
- **Robustness**: It is computationally infeasible for an adversary to produce $t$ valid signature shares such that the output of the share combining algorithm is not a valid signature.

A TSS provides the following interface:

- **Setup** $(t, n) \rightarrow (mpk, PK, SK)$ : Initializes the threshold signature scheme by generating a common public key $mpk$, a public key vector $PK = \{pk_0, ..., pk_{N-1}\}$ and a vector of private keys $SK = \{sk_0, ..., sk_{N-1}\}$, for the threshold parameters provided.
- **SigShare** $(m, sk_i) \rightarrow \sigma_i$ : Deterministically generates a signature share $\sigma_i$ for $m$, given a secret key $sk_i$.
- **VerifyShare** $(m, \sigma_i, pk_i) \rightarrow \{true, false\}$ : Verifies the validity of a signature share $\sigma_i$ for $m$, using the corresponding public key $pk_i$.

- **Combine** $(m, \{\sigma_i\}_{i \in S}, PK) \rightarrow \{\sigma, \perp\}$ : Combines $t$ signature shares $\sigma_i$ for $m$ into a threshold signature $\sigma$, or $\perp$ if the signature shares vector contained invalid or duplicate elements.
- **Verify** $(m, \sigma, mpk) \rightarrow \{true, false\}$ : Verifies the validity of the threshold signature $\sigma$ for $m$, given a common public key $mpk$.

In our protocol implementation, we use an efficient threshold scheme [57] based on bilinear groups and Gap Diffie-Hellman (GDH) assumption, to realize a common coin for the ABA protocol as well as a cryptographic proof for the verifiable broadcast primitive used. As mentioned in Section 4.2 a trusted third party is responsible for generating and distributing TSS keys. It invokes `TSS.Setup(`$f + 1, N$`)` in order to generate the TSS keys and proceeds to distribute a $(mpk, PK, sk_i)$ tuple across all replicas, where $i$ corresponds to each replica's identifier. Note that whenever possible we omit the TSS-key attributes from the method invocations in our pseudocode.

### 4.3.3 Threshold Coin-Tossing

As shown by Cachin et al. [37], a common coin can be directly realized from a threshold signature scheme by signing a unique bit string, corresponding to the name of the coin, and combining the signature shares to generate a random seed. Following the properties of a $(t, n)$-TSS, a threshold coin tossing protocol provides the following additional guarantees:

- **Unpredictability**: It is infeasible for the adversary to reliably predict the outcome of the protocol without participation from correct replica.

Our common coin protocol, illustrated in Algorithm 4.1, requires the exchange of $\mathcal{O}(N^2)$ messages, incurs a total communication cost of $\mathcal{O}(\lambda N^2)$, where $\lambda$ corresponds to the size of a threshold signature share, and terminates in a single asynchronous round.

In Alea-BFT we rely on a distributed coin tossing protocol as a common source of randomness for the ABA protocol described latter in this section.

### 4.3.4 Consistent Broadcast

Consistent Broadcast (CBC) is a protocol to deliver a payload message from a distinguished sender to all replicas. It provides no guarantees that all correct replica processes deliver the broadcast value, when the sender is faulty, however it ensures that no two correct processes deliver conflicting messages. More formally, a CBC protocol ensures the following properties [58]:

- **Validity:** If a correct sender broadcasts $m$, then all correct parties eventually deliver $m$.
- **Consistency:** If a correct party delivers $m$ and another party delivers $m'$, then $m = m'$.

**Algorithm 4.1** Distributed coin tossing (for process $P_i$)

```
1: constants:
2:     f                                  ▷ The maximum number of corrupted processes

3: state variables:
4:     Σ_i ← ∅                            ▷ The set of TSS shares received.

5: procedure COIN(C)
6:     σ_i ← TSS.SigShare(C)
7:     multicast ⟨COIN, C, i, σ_i⟩

8: upon receiving a ⟨COIN, C, j, σ_j⟩ from P_j for the first time do
9:     if TSS.VerifyShare(C, σ_j) then
10:        Σ_i ← Σ_i ∪ {σ_j}
11:        if |Σ_i| = f + 1 then
12:            σ ← TSS.Combine(C, Σ_i)
13:            output σ % 2
```

- **Integrity:** Every correct party delivers at most one request. Additionally, if the sender if correct, then the request was previously broadcast by it.

This primitive can be instantiated via the echo broadcast protocol [58]. Despite not using CBC directly in Alea-BFT, we present it as a reference point for a similar primitive based on it, which we will proceed do describe next.

### 4.3.5 Verifiable Consistent Broadcast

A Verifiable Consistent Broadcast (VCBC) protocol provides an extension of CBC that allows any party $P_i$, that has delivered the payload message $m$, to inform another party $P_j$ about the outcome of the broadcast execution, such that it can deliver $m$ immediately and terminate the corresponding VCBC instance. More formally, a VCBC protocol guarantees the following additional properties over CBC [56]:

- **Verifiability:** If a correct party delivers a message $m$, then it can produce a single protocol message $M$ that it may send to other parties such that any correct party that receives $M$ can safely deliver $m$.

- **Succinctness:** The size of the proof $\sigma$ carried by $M$ is independent of the length of $m$.

A protocol implementing VCBC is presented in Algorithm 4.2 [56], it consists of a slightly modified version of the previously presented echo broadcast protocol, using the non-interactive threshold signature scheme described in Section 4.3.2 to generate and validate a proof $\sigma$ associated with $M$. This allows the message size to be kept constant, ensuring succinctness. The message complexity of the

VCBC protocol presented is $\mathcal{O}(N)$ and its communication complexity is $\mathcal{O}(N(|m| + \lambda))$, assuming the size of a threshold signature and share is at most $\lambda$ bits.

**Algorithm 4.2** Verifiable Echo Broadcast (for process $P_i$ and tag *ID*)

**state variables:**
    $\overline{m} \leftarrow \perp$
    $\overline{\sigma} \leftarrow \perp$
    $\Sigma \leftarrow \emptyset$

**procedure** VCBC(m)
    **broadcast** $\langle \texttt{SEND}, m \rangle$

**upon** receiving a valid $\langle \texttt{SEND}, m \rangle$ message from $P_j$ **do**
    **if** $\overline{m} = \perp$ **then**
        $\overline{m} \leftarrow m$
        $\sigma_i \leftarrow TSS.SigShare(ID||H(m))$
        **send** $\langle \texttt{ECHO}, H(m), \sigma_i \rangle$ to $P_j$

**upon** receiving a valid $\langle \texttt{ECHO}, d, \sigma_j \rangle$ message from $P_j$ for the first time **do**
    **if** $TSS.VerifyShare(d, \sigma_j)$ **then**
        $\Sigma \leftarrow \Sigma \cup \{\sigma_j\}$
        **if** $|\Sigma| = \lceil \frac{N+f+1}{2} \rceil$ **then**
            $\sigma \leftarrow$ TSS.Combine($ID||d$, $\Sigma$)
            **broadcast** $\langle \texttt{FINAL}, d, \sigma \rangle$

**upon** receiving a valid $\langle \texttt{FINAL}, d, \sigma \rangle$ message from $P_i$ **do**
    **if** $H(\overline{m}) = d$ **and** TSS.Verify($ID||d$, $\sigma$) **then**
        $\overline{\sigma} \leftarrow \sigma$
        **output** $\overline{m}$

**upon** receiving a valid $\langle \texttt{REQ} \rangle$ message from $P_j$ **do**
    **if** $\overline{\sigma} \neq \perp$ **then**
        **send** $\langle \texttt{ANS}, \overline{m}, \overline{\sigma} \rangle$ to $P_j$

**upon** receiving a valid $\langle \texttt{ANS}, m, \sigma \rangle$ message from $P_j$ **do**
    **if** $\overline{\sigma} \neq \perp$ **and** TSS.Verify($ID||H(m)$, $\sigma$) **then**
        $\overline{\sigma} \leftarrow \sigma$
        $\overline{m} \leftarrow m$
        **output** $\overline{m}$

In Alea-BFT, replicas rely on VCBC to broadcast their proposals, while piggybacking an incremental sequence number in the *ID* field. Additionally, we leverage the verifiability property of VCBC to ensure all correct replicas are able actively fetch specific proposals from each other, by triggering a fallback mechanism when the proposer is faulty or the network is slow.

### 4.3.6 Asynchronous Binary Agreement

A binary agreement primitive allows correct processes to agree on the value of a single bit. Each process $P_i$ proposes a binary value $b_i \in \{0, 1\}$ and decides for a common value $b$ from the set of proposals by correct processes. Formally, a binary agreement primitive can be defined by the following properties:

- **Agreement:** If any correct process decides $b$ and another correct process delivers $b'$, then $b = b'$.
- **Termination:** Every correct process eventually decides.
- **Validity:** If all correct processes propose $b$, then any correct process that decides must decide $b$.

Following the FLP impossibility result [5], there is no deterministic algorithm capable of satisfying all the previous proprieties in the asynchronous model of Alea-BFT. A solution to this is problem is resort to a randomized model that guarantees termination in a probabilistic way. As a result, the termination property is replaced with the following:

- **Termination:** Every correct process eventually decides with probability $1$.

We instantiate this primitive via the Cobalt ABA [59] protocol, presented in Algorithm 4.3, a modified version of the protocol by Mostefaoui et al. [38] to include a fix for a liveness issue present in the original protocol. It provides optimal resilience, $\mathcal{O}(N^2)$ message complexity, and terminates in $\mathcal{O}(1)$ expected time. The common coin is instantiated using the TSS based protocol of Section 4.3.3 and dominates the communication complexity of our ABA instantiation.

In Alea-BFT, ABA is used in the agreement stage in order to determine on whether or not to deliver a proposal for a given agreement round.

## 4.4 Alea-BFT

In Alea-BFT we completely sidestep from an ACS based framework in favor of a novel pipelined architecture composed by broadcast and agreement components, executed in parallel, that communicate with each other by performing write and read operations over a set of $N$ priority queues. A priority queue consists of an auxiliary data structure to mediate the communication between components. We present the pseudocode for Alea-BFT in Algorithms 4.4 to 4.7. Algorithm 4.4 is responsible for initializing the shared state variables and starting the pipeline components upon a to call to the START procedure. Algorithm 4.4 details the internal operation of the priority queue data structure. Finally, Algorithms 4.6 and 4.7 describe the operation of the broadcast and agreement components, respectively.

**Algorithm 4.3** Asynchronous binary agreement (for process $P_i$)

1: **state variables:**
2: $r_i \leftarrow 0$
3: $values_{r_i} \leftarrow \emptyset$
4: $est_{r_i}$

5: **procedure** BA($proposal_i$)
6: $est_{r_i} \leftarrow proposal_i$
7: **while** true **do**
8: **broadcast** $\langle \texttt{VAL}, r_i, est_{r_i} \rangle$
9: **wait until** $values_{r_i} \neq \emptyset$ **then**
10: **broadcast** $\langle \texttt{AUX}, r_i, values_{r_i} \rangle$
11: **wait until** until $N - f$ $\langle \texttt{AUX}, r_i, x \rangle$ messages have been received, such that $val_{r_i} \subseteq values_{r_i}$ where $val_{r_i}$ is the of set values $x$ carried by these messages **then**
12: **broadcast** $\langle \texttt{CONF}, r_i, values_{r_i} \rangle$
13: **wait until** until $N - f$ $\langle \texttt{CONF}, r_i, S \rangle$ messages have been received, such that $S_{r_i} \subseteq values_{r_i}$ where $S_{r_i} = \bigcup S$ of set $S$ carried by these messages **then**
14: $s \leftarrow \texttt{COIN}(r_i)$
15: **if** $S_{r_i} = \{b\}$ **then**
16: **if** $b = s\%2$ **then**
17: **if** $\langle \texttt{FINISH}, b \rangle$ has not been sent **then**
18: **broadcast** $\langle \texttt{FINISH}, b \rangle$
19: $est_{r_i+1} \leftarrow b$
20: **else**
21: $est_{r_i+1} \leftarrow s\%2$
22: $r_i \leftarrow r_i + 1$

23: **upon** receiving a valid $\langle \texttt{VAL}, r, v \rangle$ message from $f + 1$ distinct replicas **do**
24: **if** $\langle \texttt{VAL}, r, v \rangle$ has not been sent **then**
25: **broadcast** $\langle \texttt{VAL}, r, v \rangle$

26: **upon** receiving a valid $\langle \texttt{VAL}, r, v \rangle$ message from $2f + 1$ distinct replicas **do**
27: $values_r \leftarrow values_r \cup \{v\}$

28: **upon** receiving a valid $\langle \texttt{FINISH}, v \rangle$ message from $f + 1$ distinct replicas **do**
29: **if** $\langle \texttt{FINISH}, v \rangle$ has not been sent **then**
30: **broadcast** $\langle \texttt{FINISH}, v \rangle$

31: **upon** receiving a valid $\langle \texttt{FINISH}, v \rangle$ message from $2f + 1$ distinct replicas **do**
32: **output** $v$ and **halt**

**Algorithm 4.4** Alea-BFT - Initialization (for process $P_i$)

```
1:  constants:
2:      N                                          ▷ The total number of processes
3:      f                                          ▷ The maximum number of corrupted processes

4:  state variables:
5:      S_i ← ∅                                    ▷ The set of delivered client requests
6:      queues_i ← ∅                               ▷ The local priority queue for each process

7:  procedure START
8:      queues_i[x] ← new pQueue() , ∀x ∈ [0, N[
9:      async BC-START()                           ▷ Start the broadcast component
10:     async AC-START()                           ▷ Start the agreement component
```

### 4.4.1 State

Processes maintain two state variables shared between components. The variable $S_i$ consisting of the set of all messages delivered by the protocol, is initialized as empty upon a call to the START procedure and updated during the execution of the agreement component. The variable $queues_i$, declared on line 6, contains $N$ priority queues, each one mapping to a distinct replica $P_i, \forall_i \in [0, N[$.

### 4.4.2 Upon Rules

In addition to the agreement loop, presented later in this section, the main logic of the protocol is expressed through a series of upon rules, triggered when a specific guard condition is satisfied. In order to correctly interpret the pseudocode presented next it is important to disambiguate some concurrency issues, specially when the rule in question interacts with shared state variables. First, every upon rule is executed atomically from start to finish. Second, only a single upon rule can be executed at a given moment in time, meaning there is no concurrency, and pending rules are FIFO queued until the current one finishes execution.

### 4.4.3 Validation

A message is only accepted by a correct process if it is considered to be valid. In the context of our protocol a valid message must follow the specified format and carry a proof of integrity and authentication for the sender. Furthermore, for a $\langle \texttt{FILLER}, entries \rangle$ message to be valid, it must be received in response to a `FILL-GAP` request and the $entries$ set must contain a non empty sequence of messages that instantly complete a VCBC instance upon delivery (see Section 4.3.5), sorted in ascending order by VCBC identifier and without gaps.

### 4.4.4 Priority Queue

A priority queue is a custom data structure for storing elements, sorted according to their priority values. We refer to each position in a priority queue as a slot, uniquely identified by a priority value associated with it. Only a single element can ever be inserted in a given slot, even after being removed, as the slot is permanently labeled as used and cannot store another element. There is a special slot called the head slot, that always points to the lowest priority slot whose value hasn't been removed yet. The pointer to the head slot progresses incrementally, conditioned by the insertion and removal of elements from the queue. A priority queue exposes the following attributes:

- **id** : The unique identifier of the queue (static).
- **head** : The priority associated with the head slot of the queue (dynamic).

Additionally, a priority queue provides an interface for interacting with its contents as described bellow:

- **Enqueue** $(v, s)$ : Add an element $v$ with a given priority value $s$ to the queue, ignore if the corresponding slot is not empty.
- **Dequeue** $(v)$ : Remove the specified element $v$ from the queue, if it is present.
- **Get** $(s) \rightarrow \{v, \perp\}$ : Retrieve the element $v$ contained in the slot specified by the priority $s$, or $\perp$ if the slot is empty.
- **Peek** $() \rightarrow \{v, \perp\}$ : Retrieve the element $v$ in the head slot of the queue, or $\perp$ if the slot is empty.

In Alea-BFT we leverage the proprieties of this structure, to mediate the communication between the broadcast and agreement components of the protocol pipeline. Every replica maintains $N$ priority queues, which are used to store the undelivered proposals pertaining to each replica, according to the priority value attributed to them. The pseudocode for a priority queue implementation can be found in Algorithm 4.5.

### 4.4.5 Queue Mapping Function

A queue mapping function is a function $F(r)$ that identifies the priority queue, over which the protocol should operate for a given round $r$. Informally it can be thought of as a leader election function, responsible for selecting which replicas pre-ordered proposals do operate over for any given $r$. This function can be any deterministic mapping from $\mathbb{N}$ to $i \in [0, N[$ as long as, for any given value $r$, there is are values $r' > r$ such that $F(r')$ spans over all elements in $[0, N[$, guaranteeing that a queue is always eventually revisited in a subsequent round. For our initial implementation of Alea-BFT we chose a queue mapping function $F(r) = r \,\%\, N$, which iterates over the priority queues following a round robin distribution.

**Algorithm 4.5** Priority Queue

```
 1: constants:
 2:     id

 3: state variables:
 4:     slots ← ∅
 5:     head ← 0

 6: procedure ENQUEUE(v, s)
 7:     slot ← slots[s]
 8:     if slot.state = ⊥ then
 9:         slot.v ← v
10:         slot.state ← USED

11: procedure GET(s)
12:     slot ← slots[s]
13:     if slot.state ≠ ⊥ then
14:         return slot.v
15:     return ⊥

16: procedure PEEK()
17:     return GET(head)

18: procedure DEQUEUE(v)
19:     s.state ← RM, ∀_s ∈ slots : s.v = v
20:     head ← min(∀_i ∈ head ≤ i < len(slots) : slots[i].state ≠ RM)
```

### 4.4.6 Broadcast Component

The broadcast component is responsible for establishing a local pre-order over the client updates received and propagating that order to other replicas. The overall component flow, illustrated in Algorithm 4.6, starts with replicas receiving client requests and storing them in a buffer. When the number of requests in the buffer exceeds a certain threshold $B$, corresponding to the batch size ($B = 1$ unless batching is mentioned), the requests are removed from the buffer to form a proposal. An incremental sequence number is attributed to the proposal and the pair is disseminated via a VCBC primitive. Note that a sequence number is local to each replica, meaning that proposals originating from different replicas may share the same sequence number. When a replica VCBC-delivers a proposal, it stores it in a backlog pertaining to the proposer, locally sorted according to the respective priority value, such that it can later be picked up by the next stage of the pipeline. Particularly, every replica maintains $N$ separate proposal backlogs, consisting of undelivered pre-ordered proposals by each of the $N$ replicas. Additionally, each replica maintains two local state variables, a buffer of pending requests $buf_i$, and an integer value, $priority_i$, indicating the next sequence number it should attribute to a proposal. The main logic of this component is split between two upon rules:

**Upon rule 1 (lines 9 to 15):** The first rule is triggered, by any correct process $P_i$, upon the reception of a client message $m$ to be totally ordered by the protocol. It is responsible for selecting a batch of $B$ transactions from the pending buffer, attributing a local sequence number to it and broadcasting this pre-odered proposal to all replicas. Process $P_i$ then proceeds as follows:

- If the set of delivered messages $S_i$ does not contain the client message $m$, append it to the buffer $buf_i$, or ignore it otherwise (lines 10 to 11).
- If the size of the buffer reached a threshold $B$, input $buf_i$ into a VCBC instance tagged with *ID* $(i, priority_i)$ indicating that process $P_i$ attributed the local priority value $priority_i$ to a proposal containing the buffer contents (lines 12 to 13).
- Increment the value of $priority_i$, so that it can be assigned to the next proposal from $P_i$ and clear the buffer $buf_i$. (lines 14 to 15).

**Upon rule 2 (lines 16 to 20):** The second rule is triggered, by any correct process $P_i$, upon the delivery of a proposal $m$ for a given VCBC instance tagged with *ID* $(j, priority_j)$, where $j$ corresponds to the identifier of the replica $P_j$ that proposed $m$, and $priority_j$ to the sequence number attributed to it by $P_j$. Process $P_i$ proceeds as follows:

- Insert the delivered proposal $m$ pair into the slot $priority_j$ of the priority queue $Q_j$, mapping to $P_j$. This corresponds to adding the $m$ to the backlog of $P_j$ in $P_i$, with the priority value $priority_j$ (lines 17 to 18).
- If the set $S$ contains $m$, indicating that it had already been delivered, then process $P_i$ immediately removes it from $Q_j$ in order to avoid ordering duplicate messages, effectively deleting $m$ and progressing through the backlog (lines 19 to 20).

Note that the broadcast primitive used, VCBC, does not guarantee that all replicas receive a proposal, only that no conflicting proposals are delivered. This allows for a more efficient communication, when compared to RBC, but requires an additional mechanism to handle the situation when certain processes fail to receive a proposal. We address this situation by allowing replicas to actively fetch proposals from other replicas if required, leveraging the verifiable property of VCBC. The VCBC execution can be therefore seen as a fast path for proposal dissemination, while actively fetching proposals consists of a fallback mechanism that imposes additional overhead.

### 4.4.7 Agreement Component

The agreement component, presented in Algorithm 4.7, is responsible for establishing a total order over the backlogs of replica proposals created by the execution of the broadcast component. It proceeds in rounds, such that for each round the backlog of proposals pertaining to a certain replica (which for ease of description we refer to as the round leader) is selected for processing according to a mapping function presented in Section 4.4.5. A correct replica examines the leader's proposal backlog and inputs a value

**Algorithm 4.6** Alea-BFT - Broadcast Component (for process $P_i$)

1: **constants:**
2: $B$ ▷ The batch size parameter.

3: **state variables:**
4: $buf_i$
5: $priority_i$ ▷ The local queue priority value for the process

6: **procedure** BC-START
7: $buf_i \leftarrow \emptyset$
8: $priority_i \leftarrow 0$

9: **upon** receiving a valid message $m$, from a client **do**
10: **if** $m \notin S_i$ **then**
11: $buf_i \leftarrow buf_i \cup \{m\}$
12: **if** $|buf_i| = B$ **then**
13: **input** $buf_i$ to VCBC $(i, priority_i)$ ▷ Start VCBC instance
14: $buf_i \leftarrow \emptyset$
15: $priority_i \leftarrow priority_i + 1$

16: **upon** outputting $m$ for VCBC $(j, priority_j)$ **do**
17: $Q_j \leftarrow queues_i[j]$
18: $Q_j.Enqueue(priority_j, m)$
19: **if** $m \in S$ **then**
20: $Q_j.Dequeue(m)$

into an ABA execution depending on its contents. Particularly, if it had previously delivered a proposal from the current leader with priority $s$, it expects the backlog to contain a proposal with priority $s + 1$ to be ordered next. Or $s + 2$, if the $(s + 1)$-th leader proposal happens to be a duplicate from an already ordered proposal originating from a distinct replica. If it contains such proposal, it inputs $1$ into an ABA instance, or $0$ otherwise. The outcome of the ABA will dictate whether the pre-ordered proposal should be totally ordered for that round or not. Due to the fact that the broadcast primitive used by Alea-BFT does not guarantee totality, some replicas may need execute a fallback sub-protocol to actively fetch this value from others. Processes maintain a single state variable $r_i$, serving as an unique identifier for the current agreement round. The execution of the agreement component starts with a call to the AC-START procedure (line 3), which initializes the local variable $r_i$ to $0$ and begins executing the agreement loop.

**Agreement loop (lines 5 to 16):** Each iteration $r_i$ of the agreement loop operates over a single priority queue, deterministically selected by a queue mapping function, as described in Section 4.4.5. Let $Q_a = queues[F(r_i)]$ denote the priority queue selected in for the current round. A correct process $P_i$ proceeds as follows:

- Run an ABA instance tagged with ID $(r_i)$ to determine whether the $value$ in the head slot of $Q_a$ should be delivered for this round. Process $P_i$, inputs $1$ into the ABA if its local $Q_a$ contained a $value$ in the head slot, retrieved by invoking a PEEK operation over $Q_a$, or $0$ otherwise (lines 6 to 9).
- If the ABA execution decided for $0$, simply proceed to the next loop iteration, otherwise:
  - If process $P_i$ input $0$ into the ABA send a `FILL-GAP` message to all processes that voted for $1$. This is required because at this point in time $P_i$ is unaware of the value to deliver for $r_i$ and therefore must request it from another process (lines 12 to 13).
  - Block execution until the head slot of $Q_a$ contains a value to be delivered via a call to the AC-DELIVER procedure. The value of the head slot can be updated by the delivery by a pending VCBC instance, either through "normal" execution or as result of the reception of a `FILLER` message (line 14).

In addition to the main agreement loop, the agreement component also defines two upon rules to handle the reception of valid `FILL-GAP` and `FILLER` messages:

**Upon rule 1 (lines 17 to 21):** The first rule is triggered, by any correct process $P_i$, upon the reception of a valid $\langle \texttt{FILL-GAP}, q, s \rangle$ message from $P_j$, where $q$ identifies a priority queue $Q_q$, and $s$ specifies the current head slot of $Q_q$ in $P_j$. Process $P_i$ then proceeds as follows:

- Check if its local backlog pertaining to $P_q$ is more advanced than the one of $P_j$, by comparing the head pointer of its $Q_q$ with $s$ (line 19). If it's lower this indicates that $P_i$ cannot satisfy the `FILL-GAP` request thus ignoring it, otherwise:

– Compute and store in $entries$ a verifiable message $M$ (see Section 4.3.5), for all VCBC instances originating from $P_q$ tagged with a priority compromised between the value $s$ requested by $P_j$ and current head slot of $Q_q$ in $P_i$ (line 20).
– Send a `FILLER` message to $P_j$ containing all the VCBC verifiable messages $M$, computed in the previous step (line 21).

**Upon rule 2 (lines 22 to 24):** The second rule is triggered, by any correct process $P_i$, upon the reception of a valid $\langle \texttt{FILLER}, entries \rangle$ message. This message is received as a response to a `FILL` request and contains the required information necessary for $P_i$ to progress in the execution of the protocol, by completing pending VCBC instances, after blocking in line 14. Process $P_i$ proceeds as follows:

- Deliver all $M$ messages in $entries$ to the corresponding VCBC instances. Note that a VCBC delivery also triggers the second upon rule of the broadcast component.

Finally the AC-DELIVER procedure (line 25), called during the execution of the agreement loop, is responsible for delivering a totally ordered message $m$ to the application layer (line 29). Additionally, this procedure also removes $m$ from all priority queues and appends it to the set of delivered requests $S$.

**Algorithm 4.7** Alea-BFT - Agreement Component (for process $P_i$)

**state variables:**
  $r_i$ ▷ The current agreement instance

**procedure** AC-START
  $r_i \leftarrow 0$
  **while** true **do**
    $Q \leftarrow queues_i[F(r_i)]$
    $value \leftarrow Q.Peek()$
    $proposal \leftarrow v \neq \perp$ ? $1 : 0$
    **input** $proposal$ to ABA $(r_i)$ ▷ Start binary agreement instance
    **wait until** ABA $(r_i)$ delivers $b$ **then**
      **if** $b = 1$ **then**
        **if** $Q.Peek() = \perp$ **then**
          broadcast $\langle$FILL-GAP$, Q.id, Q.head\rangle$
        **wait until** $(v \leftarrow Q.Peek()) \neq \perp$ **then**
          AC-DELIVER$(v)$
    $r_i \leftarrow r_i + 1$ ▷ Progress to the next round

**upon** receiving a valid $\langle$FILL-GAP$, q, s\rangle$ message from $P_j$ **do**
  $Q \leftarrow queues_i[q]$
  **if** $Q.head \geq s$ **then**
    $entries \leftarrow$VCBC$(queue, s')$.REQ $\forall_{s'} \in [s, Q.head]$
    **send** $\langle$FILLER$, entries\rangle$ to $P_j$

**upon** delivering a valid $\langle$FILLER$, entries\rangle$ message **do**
  **for each** $\langle$ANS$, *, *\rangle$ message $M \in entries$ **do**
    **deliver** $M$ to the corresponding VCBC

**procedure** AC-DELIVER(value)
  **for each** $n \in N$ **do**
    $queues_i[n].Dequeue(value)$
  $S_i \leftarrow S_i \cup \{value\}$
  **output** value ▷ Deliver to the application layer

## 4.5 Efficiency Analysis

In this section we analyse theoretical efficiency of the Alea-BFT protocol, according to time, message and communication complexity measures, which are summarized in Table 4.1.

By analysing Alea-BFT we observe that, for every particular proposal payload to be delivered, message exchanges occur in three different places. First, during the execution of the broadcast component, a replica initiates a VCBC instance to disseminate the pre-ordered proposal across all replicas, which are queued in a priority queue slot according to the priority value assigned to it. Second, all replicas participate in successive ABA instances in order to decide, whether or not, to deliver a particular slot's contents. We denote by $\sigma$ the average number of ABA instances executed over a single slot before it decides for $1$ and its contents are scheduled for delivery. Finally, a fallback sub-protocol is triggered by replicas that did not VCBC-deliver the proposal before the corresponding ABA execution decided for $1$, in order to actively fetch it from the other replicas.

### 4.5.1 Time Complexity

Time (round) complexity is defined as the expected number of communication rounds before a protocol terminates, or before a given value is output in case of a continuous protocol with on-line inputs and outputs, such as Alea-BFT (atomic broadcast). The first and third steps terminate in constant time $\mathcal{O}(1)$, whereas the total number of rounds required for the agreement component to decide depend on the value of $\sigma$, therefore bounding the overall time complexity of Alea-BFT as $\mathcal{O}(\sigma)$.

### 4.5.2 Message Complexity

Message complexity is defined as the expected number of messages generated by correct replicas during the execution of the protocol. The VCBC instance executed during the broadcast phase generates $\mathcal{O}(N)$ messages, every ABA instance exchanges $\mathcal{O}(N^2)$ messages and finally the third recovery phase incurs an overhead of $\mathcal{O}(N)$ messages per each replica that triggers this fallback protocol. Hence, the message complexity of Alea-BFT is $\mathcal{O}(\sigma N^2)$, due to the $\sigma$ ABA instances that are executed per priority queue slot prior to delivery.

### 4.5.3 Communication Complexity

Communication complexity consists of the expected total bit-length of messages generated by correct replicas during the protocol execution. Let $|m|$ correspond to the average proposal size and $\lambda$ the size of a threshold signature share. The execution of VCBC incurs a communication complexity of $\mathcal{O}(N(|m| + \lambda))$, each ABA instance requires correct nodes to exchange $\mathcal{O}(\lambda N^2)$ bits and finally each

replica that triggers the recovery phase adds an additional communication cost of $\mathcal{O}(N(|m| + \lambda))$ bits. This results in an expected total communication complexity of $\mathcal{O}(\sigma\lambda N^2 + N^2(|m| + \lambda))$, due to the $\sigma$ ABA executions and up to $N$ recovery round triggers.

**Table 4.1:** Complexity of Alea-BFT decomposed by stages.

| **Stage** | Message | Communication | Time |
|---|---|---|---|
| Broadcast | $\mathcal{O}(N)$ | $\mathcal{O}(N(\|m\| + \lambda))$ | $\mathcal{O}(1)$ |
| Agreement | $\mathcal{O}(\sigma N^2)$ | $\mathcal{O}(\sigma\lambda N^2)$ | $\mathcal{O}(\sigma)$ |
| Recovery | $\mathcal{O}(N^2)$ | $\mathcal{O}(N^2(\|m\| + \lambda))$ | $\mathcal{O}(1)$ |
| **Total** | $\mathcal{O}(\sigma N^2)$ | $\mathcal{O}(\sigma\lambda N^2 + N^2(\|m\| + \lambda))$ | $\mathcal{O}(\sigma)$ |

### 4.5.4 Quantifying Sigma ($\sigma$):

As previously mentioned, Alea-BFT does not provide a constant time reduction from VCBC and ABA to atomic broadcast. In particular, this is because multiple redundant zero deciding ABA instances may be executed over the same priority queue slot until its contents are considered to be totally ordered. However, we argue that despite being theoretically unbounded, the value of $\sigma$ is in practice a very small constant, which ultimately converges to the optimal value of 1. This statement comes from the observation that given a round-robin queue mapping function $F$, the same queue is revisited every $N$ epochs, meaning $N$ sequential ABA instances must have been executed by the time a particular queue is revisited. Considering the validity property of ABA, which states that the decided value must have been proposed by a correct process, then the termination of a VCBC instance by $N - f$ correct replicas guarantees that the next ABA execution pertaining to it will decide for $1$. Therefore, in order for the average value of $\sigma$ to increase by a single unit, replicas must complete $N$ sequential ABA executions for every single VCBC instance, a very unlikely scenario given the characteristics of these protocols.

In Chapter 6 we present practical results that further corroborate the hypothesis that the expected value of $\sigma$ is constant.

## 4.6 Correctness Proof

In this section, we present a correctness proof for the Alea-BFT protocol. It is organized in four subsections, one for each property of the algorithm - validity, agreement, integrity and total order. All proofs are written according to the structured proof format by Lamport [60] and assume that $N = 3f + 1$. For convenience, we start by presenting some auxiliary definitions, which will be in effect for all the proofs presented in this section.

**Definition 1** *If a correct process $P_i$ enters round $r$, let $S_r^{(i)}$ denote the set of requests delivered by $P_i$ by the time it enters $r$. We say that consensus holds on entry to round $r$ if for any two correct processes $P_i$ and $P_j$ that enter $r$, then $S_r^{(i)} = S_r^{(j)}$. If consensus holds on entry to round $r$, and any correct process does enter round $r$, we denote by $S_r$ the common value of the $S_r^{(*)}$.*

It is trivial to see that consensus always holds for round $0$, considering $S_0$ starts as empty. For this reason by proving the agreement and total order properties for a round where consensus holds, we can recursively generalise the results to every other round, using $r = 0$ as a starting point.

**Definition 2** *A correct process $P_i$ is said to be $\texttt{Prepared}(i, s)$ for $m$ on round $r$ when it has delivered all VCBC $(i, s')$ instances tagged with a priority $s' \leq s$, where $s$ corresponds to the lowest priority assigned by $P_i$ to a message $m \notin S_r^{(i)}$. The value of $m$ corresponds to the result of VCBC $(i, s)$.*

Note that stating that a process is $\texttt{Prepared}(i, s)$ for $m$, is the equivalent of invoking the `Peek` operation over the priority queue $Q_i$ and obtaining the value $m \neq \perp$, located in its head slot of index $s$.

### 4.6.1 Agreement

The proof of agreement is structured as Lemma 1 and Theorem 1. Lemma 1 is used to prove agreement on the value of $\texttt{Prepared}(i, -)$ for any agreement round where consensus holds at entry and Theorem 1 uses the previous lemma as support to conclude the reasoning.

**Lemma 1** *If some correct process has $\texttt{Prepared}(i, -)$ for value $m$ on round $r$, then no other correct process can have $\texttt{Prepared}(i, -)$ for $m'$, on the same agreement round, where consensus holds at entry, such that $m \neq m'$.*

For this proof we assume that two correct processes $P$ and $P'$ are $\texttt{Prepared}(i, -)$ for values $m$ and $m'$ respectively, on round $r$ where consensus holds at entry, and proceed to demonstrate that $m \neq m'$ leads to a contradiction due to a violation the consistency property of VCBC.

1. ASSUME: 1. Both $P$ and $P'$ are correct processes.
   2. Process $P$ is $\texttt{Prepared}(i, s)$ for $m$ on round $r$.

3. Process $P'$ is $\texttt{Prepared}(i, s')$ for $m'$ on round $r$.
4. Consensus holds for $r$.

PROVE: $m = m'$.

2. Process $P$ delivered $m$ for VCBC $(i, s)$.

PROOF: This follows directly from assumptions 1 and 2, and the definition of the `Prepared` predicate presented in Definition 2.

3. Process $P'$ delivered $m'$ for VCBC $(i, s')$.

PROOF: The same argument of 2 applies here.

4. $s = s'$.

PROOF: By assumptions 2-4 and Definition 2.

5. Q.E.D.

PROOF: Step 4 implies that, if $m \neq m'$, then the correct processes $P$ and $P'$ must have delivered different values for the same VCBC instance tagged with $(i, s)$. This contradicts the consistency property of VCBC proving the lemma.

**Theorem 1 (Agreement)** *If a correct process delivers a message $m$, then all correct processes eventually deliver $m$.*

Let us assume, without loss of generality, that some correct process $P$ has delivered a message $m$, during a given round $r$ for which consensus holds at entry. Based on the assumption, it suffices to demonstrate, without loss of generality, that a second correct process $P'$, which is yet to deliver $m$, eventually does so.

1. ASSUME: 1. Both $P$ and $P'$ are correct processes.
2. Process $P$ invoked $\texttt{DELIVER}(m)$ for round $r$.
3. Consensus holds for $r$.

PROVE: Process $P'$ invokes $\texttt{DELIVER}(m)$.

2. Process $P'$ decides $1$ for ABA $(r)$.

2.1. Process $P$ decided $1$ for ABA $(r)$.

PROOF: By assumptions 1 and 2, that $P$ is correct and follows the protocol rules, therefore only delivering a message for any round $r$ if the corresponding ABA execution decided for $1$.

2.2. Q.E.D.

PROOF: By 2.1 and the agreement and termination properties of ABA.

We now have two cases to consider. One where process $P'$ is `Prepared`$(F(r), s)$ for some value $m'$ at the beginning of round $r$ and one where it hasn't prepared yet. Note that $F(r)$ corresponds to the results of the queue mapping function for round $r$.

3. CASE: Process $P'$ is `Prepared`$(F(r), s)$ for $m'$.

   3.1. $m = m'$.

   PROOF: By assumption 3 and Lemma 1.

   3.2. Q.E.D.

   PROOF: By 3.1, and the assumption that $P'$ is correct, process $P'$ delivers $m$ for round $r$.

4. CASE: Process $P'$ is not `Prepared`$(F(r), s)$.

   4.1. Process $P'$ broadcasts a $\langle$`FILL-GAP`$, F(r), -\rangle$ request.

   PROOF: By 2 and 4. Any correct process that hasn't `Prepared`$(F(r), s)$ upon a ABA $(r)$ decision for $1$ initiates a recovery sub routine, which starts with the broadcast of a `FILL-GAP` request for the queue $F(r)$.

   4.2. Process $P'$ receives a valid $\langle$`FILLER`$, -\rangle$ reply.

      4.2.1. At least one correct process $P_c$ input $1$ to ABA $(r)$.

      PROOF: By 2 and the validity property of ABA.

      4.2.2. Process $P_c$ was `Prepared`$(F(r), s)$ at the start of round $r$.

      PROOF: By 4.2.1 and the assumption that $P_c$ is correct.

      4.2.3. Q.E.D.

      PROOF: By 4.2.2 and definition 2, the correct process $P_c$ must have VCBC delivered all proposals from $P_{F(r)}$ tagged with a priority value $s' \leq s$. Therefore, by the verifiability of VCBC, process $P_c$, can produce a `FILLER` message that completes all VCBC instances whose result $P'$ is unaware.

   4.3. Process $P'$ becomes `Prepared`$(F(r), s)$ for $m'$.

   PROOF: By 4.2 and the verifiability property of VCBC.

   4.4. Q.E.D.

PROOF: By 4.3, the same argument of 3 applies here.

### 4.6.2 Integrity

**Theorem 2 (Integrity)** *Every correct process delivers any message $m$ at most once.*

For this proof we assume, without loss of generality, that a correct process $P_i$ has delivered a message $m$ for a given agreement round $r$, and proceed to demonstrate that it cannot deliver $m$ for any subsequent round due to its inability to become $\texttt{Prepared}(-,-)$ for $m$, which is a prerequisite for delivery according to the protocol spec.

1. ASSUME: 1. Process $P_i$ is correct.
   2. Process $P_i$ invoked $\texttt{DELIVER}(m)$ for round $r$.

   PROVE: Process $P_i$ cannot deliver $m$ for any round $r' > r$.

2. $S_{r+1}^{(i)} = S_r^{(i)} + \{m\}$.

   PROOF: By assumptions 1 and 2, process $P_i$ is correct and has delivered $m$ for round $r$. Therefore, updating its delivered set for the next round to include $m$.

3. Process $P_i$ cannot have $\texttt{Prepared}(F(r'),-)$ for value $m$, for any round $r' > r$.

   PROOF: By 2 and Definition 2, it is impossible for a correct process $P_i$ to prepare for any value $m$ during $r'$ if $m \in S_{r'}^{(i)}$.

4. Q.E.D.

   PROOF: To deliver $m$ for a round $r'$, the correct process $P_i$ must have $\texttt{Prepare}(F(r'),-)$ for the value $m$ during $r'$. By 3, this is impossible if $P_i$ has delivered $m$ for a prior round since $m \in S_{r'}^{(i)}$, thereby proving the theorem.

### 4.6.3 Validity

The proof of validity is structured as Lemma 2 and Theorem 3. Lemma 2 formalizes an upper bound on the number of agreement rounds required to deliver a message when certain preconditions are met. Theorem 3 builds upon this lemma to conclude the proof.

**Lemma 2** *If $2f+1$ correct processes have $\texttt{Prepared}(i,s)$ for value $m$ by the time they enter round $r$, then $m$ is guaranteed to be delivered by the next round $r' \geq r$, such that $F(r') = i$.*

1. ASSUME: 1. Process $P$ is correct.
   2. At least $2f+1$ correct processes are $\texttt{Prepared}(i,s)$ for $m$ by round $r$.

3. $r' \geq r : F(r') = i$.

4. Consensus holds for $r$ and $r'$.

PROVE: Process $P$ must have delivered $m$ by the end of round $r'$.

2. At least $2f + 1$ processes input 1 into ABA $(r')$.

PROOF: This follows from assumption 2, that $2f + 1$ processes have $\texttt{Prepared}(i, s)$ on entry to round $r$, therefore proposing $1$ into the next ABA execution pertaining to queue $i$, which by assumption 3 corresponds to $r'$.

3. Process $P$ decides $1$ for ABA $(r')$.

PROOF: By 2, and the validity and termination properties of ABA.

4. Q.E.D.

PROOF: The same reasoning from the steps 3 and 4 from Theorem 1 can be applied here.

**Theorem 3 (Validity)** *If a correct process broadcasts a message $m$, then some correct process eventually delivers $m$.*

1. ASSUME: 1. Both $P$ and $P_i$ are correct processes.

2. Process $P_i$ invoked $\texttt{SEND}(m)$.

PROVE: Process $P$ invokes $\texttt{DELIVER}(m)$

2. Every correct process delivers $m$ for VCBC $(i, s)$ .

PROOF: Let $s$ denote the priority value attributed by $P_i$ to $m$. Then, by the assumption that $P_i$ is correct and the validity property of VCBC all correct processes deliver $m$ tagged with $(i, s)$.

3. All correct processes become $\texttt{Prepared}(i, s)$ for $m$.

PROOF: This follows directly from 2.

4. Q.E.D.

PROOF: By 3, which fulfills the pre-requirements for Lemma 2 to hold.

### 4.6.4 Total Order

**Theorem 4 (Total Order)** *If two correct processes deliver messages $m$ and $m'$ then both processes deliver $m$ and $m'$ in the same order.*

For this proof, we assume without loss of generality that two correct processes $P$ and $P'$ have respec-

tively delivered $m$ and $m'$, for round $r$ where consensus holds on entry. Based on this assumption it suffices to prove that $m = m'$, in order to demonstrate total order.

1. ASSUME: 1. Both $P$ and $P'$ are correct processes.
   2. Process $P$ invoked $\texttt{DELIVER}(m)$ for round $r$.
   3. Process $P$ invoked $\texttt{DELIVER}(m')$ for round $r$.
   4. Consensus holds for $r$.
   5. $F(r) = i$.

   PROVE: $m = m'$.

2. Process $P$ has $\texttt{Prepared}(i, s)$ for $m$ during round $r$.

   PROOF: This follows from the assumption the $P$ is correct and has invoked $\texttt{DELIVER}(m)$ for round $r$, since preparing for a value always precedes its delivery.

3. Process $P'$ has $\texttt{Prepared}(i, s)$ for $m'$ during round $r$.

   PROOF: The same argument as 2 applies here.

4. Q.E.D.

   PROOF: By 2 and 3, and Lemma 1.

# 5

# Implementation

## Contents

In this chapter we present our Java implementation of Alea-BFT and complementary protocols, available at https://github.com/nosofa/ao-master-thesis. Section 5.1 describes the baseline structure of our implementation. Section 5.2 presents an overview of the particular implementation of each (sub) protocol. Finally, in Section 5.3 we provide a description of the benchmark application which was used for the experimental evaluation of Chapter 6.

## 5.1 Baseline

Our implementation is organized according to a class hierarchy, where all (sub) protocols extend an abstract class `Protocol` parameterizable by two generic types `I` and `O` corresponding to the input and output types of the protocol. Following the structure presented in Figure 4.2, specific `Protocol` implementations can be organized in a modular manner, meaning an outer protocol can provide input and route messages into sub protocols while simultaneously being responsible for handling messages and outputs produced by them.

**Listing 5.1:** Protocol abstract class.

```
/**
 * A protocol that handles input and message events.
 *
 * @param <I> the protocol input type.
 * @param <O> the protocol output type.
 */
public abstract class Protocol<I, O> {

    private final String pid;

    // omitted constructor...

    public String getPid() {
        return pid;
    }

    public abstract Step<O> handleInput(I input);
    public abstract Step<O> handleMessage(ProtocolMessage message);
}
```

Every (sub) protocol in our implementation is represented by an instance of `Protocol` uniquely identifiable by its protocol identifier `pid`, accessible via the `getPid()` method. Protocol instances are completely passive and simply react to outside events, updating their internal state in between events and generating protocol steps. The `Protocol` class provides two abstract methods to signal outside events that should be implemented in order to encode specific protocol logic. The `input(I input)` method can be invoked to signal an input event, such as a boolean vote for a binary agreement execution or a value to be dispersed in a broadcast protocol. The reception of a message can signaled by the invocation of the `handleMessage(ProtocolMessage message)` method. All messages extend an abstract `ProtocolMessage` characterized by a `pid` uniquely identifying the protocol instance to where it should be routed, a `type` parameter indicating the particular message type inside the protocol and a `sender` field corresponding to the replica that generated the message.

**Listing 5.2:** Protocol Message abstract class.

```
/**
 * A generic protocol message.
 */
public class ProtocolMessage {

    private final String pid;
    private final Integer type;
    private final Integer sender;

    public ProtocolMessage(String pid, Integer type, Integer sender) {
        this.pid = pid;
        this.type = type;
        this.sender = sender;
    }

    // ...
}
```

We refer to the state updates produced by processing these events as protocol steps, represented by the `Step<O>` class in the context of our implementation. It contains all the response messages generated during this protocol step as well as any output values, as indicated by the generic type `O`. In a protocol step a general a `ProtocolMessage` is wrapped as the payload of a `TargetedMessage` class which contains additional information regarding the target recipients of the original message.

**Listing 5.3:** Protocol Step class.

```
import java.util.List;

/**
 * Each time input (protocol input or incoming messages)
 * is provided to a protocol instance, a step is produced
 * potentially containing output values and network messages.
 *
 * @param <O> the protocol output type.
 */
public class Step<O> {

    private List<TargetedMessage> messages;
    private List<O> output;

    public Step(List<TargetedMessage> messages, List<O> output) {
        this.messages = messages;
        this.output = output;
    }

    // ...
}
```

As previously mentioned, the task of processing a protocol step is delegated to the outer protocols, and ultimately to an application layer that sits on top of the protocol stack. In addition to implementing specific application logic over the protocol output this uppermost layer is also responsible for specifying and managing the communication interfaces used by the protocols while routing the messages generated by them to the specified targets, using a `Transport` interface.

**Listing 5.4:** Transport interface.

```
/**
 * The transport used to control messaging between replicas
 * as well as clients.
 *
 * @param <T> the encoded message type
 */
public interface Transport<T> {
```

```
    void sendToReplica(int replicaId, T data);
    void sendToClient(int clientId, T data);
    void multicast(T data, int... replicas);
}
```

We provide an implementation of the transport interface `TcpTransport` which implements reliable point-to-point links using Transport Control Protocol (TCP) sockets, optionally authenticated using Hash-Based Message Authentication Code (HMAC) with SHA-256 and 32 byte keys shared by each replica pair. As the `Transport` interface is parameterizable with a transmissible type `T` (a byte array in the `TcpTransport` implementation provided) it is also required to implement an encoder based on the `MessageEncoder` interface.

**Listing 5.5:** Message Encoder interface.

```
/**
 * A component that transforms messages
 * to and from a transmissible format.
 *
 * @param <T> the common transmissible type
 */
public interface MessageEncoder<T> {
    T encode(ProtocolMessage message);
    ProtocolMessage decode(T data);
}
```

The `MessageEncoder` interface provides methods for conversion of a `ProtocolMessage` to and from a transmissible message format `T` matching the one from the `Transport` implementation being used.

## 5.2 Protocols

In this section we provide an overview of the protocols implemented in the scope of this thesis. Particularly, we provide implementations of Alea-BFT, HoneyBadger, Dumbo and Dumbo2 as well as all the underlying sub protocols required by them.

As explained in Section 5.2 all (sub) protocols extend the same parameterizable abstract class `Protocol`. Concrete implementations of the same primitive (e.g., binary agreement), despite encoding distinct event handling logic, usually share common parameterizations for the `Protocol` input and

output types. For better organization and modularity we group together different instantiations of the same primitive under a common parameterization of `Protocol` which can optionally also extend its functionality. In Figure 5.1 we present a diagram illustrating the class structure for all primitives used in Alea-BFT.

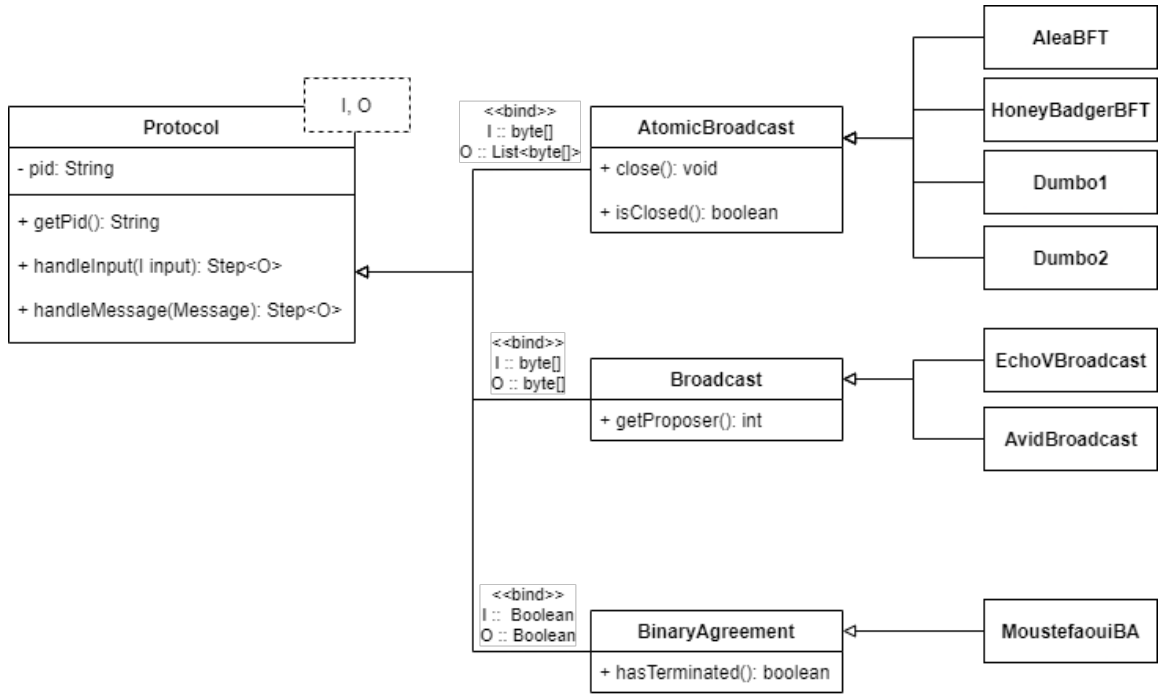



**Figure 5.1:** Class structure in our implementation.

The implementations of Alea-BFT, HoneyBadger, Dumbo and Dumbo2 all extend an `AtomicBroadcast` class which is itself a child of the baseline abstract `Protocol` class, parameterized with a byte array as input and a stream of byte arrays as output. Since atomic broadcast protocols are modeled as a channel with on-line inputs and outputs, contrary to the other one shot instance protocols in this section, the `AtomicBroadcast` class provides two extra methods to manage the channel state. When an outer layer protocol/application has determined that it is ready to close the channel, it may invoke `close()` signaling its termination intent but must continue to process messages until `isClosed()` returns true.

The `Broadcast` class groups together all broadcast sub protocols, offering a parametrization of `Protocol` with a byte array as both the input and the output. Additionally, it provides a `getProposer()` method which can be used to retrieve the unique identifier associated with the replica that initiated the broadcast. Note that for verifiable broadcast protocols, used by Alea-BFT and Dumbo2, which require an additional proof to be output, both the broadcast value and the proof are encoded into the output byte array.

Finally, concrete binary agreement primitives implement the `BinaryAgreement` class, parameterized by a boolean input indicating each replica's initial preference as well as a boolean output corresponding

to the final decision. The `BinaryAgreement` class also provides a `hasTerminated()` method allowing us to retrieve information regarding the internal state of the protocol being executed. As in binary agreement protocols, sometimes, a value output doesn't necessarily indicate the protocol has terminated. In the context of this work we only provide a single `BinaryAgreement` implementation of the ABA protocol in Cobalt [59] but our modular architecture makes its replacement trivial.

Note that we omitted a description of specific sub protocols that do not share common primitives with Alea-BFT but ,nonetheless, where implemented in the context of this work for experimental evaluation. One such example is the MVBA protocol, which is used as building block for Dumbo2.

## 5.3 Application

In Section 5.1, we mentioned the requirement for an application layer that sits on top of a protocol stack and is responsible amongst other tasks for the managing application logic based on the outputs of its inner protocols. We provide a single application layer implementation `BenchmarkReplica` which was used to perform the benchmarks presented in Chapter 6.

**Listing 5.6:** Benchmark Replica class.

```
// BenchmarkReplica.java
import java.util.List;

public class BenchmarkReplica {

    private final AtomicBroadcast protocol;
    private final Transport<byte[]> transport;
    private final MessageEncoder<byte[]> encoder;

    private final Benchmark benchmark = new Benchmark();

    public BenchmarkReplica(
            AtomicBroadcast protocol,
            Transport<byte[]> transport,
            MessageEncoder<byte[]> encoder
    ) {
        this.protocol = protocol;
        this.transport = transport;
        this.encoder = encoder;
    }
```

```
    public void input(byte[] tx) {
        benchmark.logInput(tx);
        Step<List<byte[]>> step = protocol.handleInput(tx);
        handleStep(step);
    }

    public void handleMessage(byte[] encoded) {
        ProtocolMessage message = encoder.decode(encoded);
        benchmark.logRecvMsg(message);
        Step<List<byte[]>> step = protocol.handleMessage(message);
        handleStep(step);
    }

    private void handleStep(Step<List<byte[]>> step) {
        for (TargetedMessage message: step.getMessages()) {
            benchmark.logSentMsg(message);
            byte[] encoded = encoder.encode(message.getContent());
            for (Integer target: message.getTargets())
                transport.sendToReplica(target, encoded);
        }

        for (List<byte[]> txs: step.getOutput()) {
            benchmark.logOutput(txs);
        }
    }

    public Benchmark getResults() {
        return benchmark;
    }
}
```

It takes an `AtomicBroadcast` implementation, as well as a `Transport` layer and corresponding `MessageEncoder` as constructor parameters. This allows us to execute benchmarks over different atomic broadcast protocols simply by replacing the concrete implementation that is passed into the constructor. At the core of a benchmark replica is the `Benchmark` attribute, which is responsible for logging all events that we deemed relevant for our benchmark metrics. In particular, it keeps track of in-

puts into the atomic broadcast protocol, reception and dispatch of messages and delivery of totally ordered requests. Both `input(byte[] tx)` and `handleMessage(byte[] encoded)` are triggered by the `BenchmarkReplica` to signal external events, generating protocol steps that are handled by the `handleStep(Step<List<byte[]>> step)` method, responsible for forwarding any messages generated as well as logging the outputs. Finally, benchmark results can be retrieved by calling the `getResults()` method, which returns a `Benchmark` object containing an event log that can later be parsed and analysed.

# 6

# Evaluation

## Contents

In this chapter we present an experimental evaluation of the Alea-BFT against HoneyBadgerBFT and both Dumbo protocols in a Wide Area Nework (WAN). Section 6.1 describes the evaluation environment including hardware specifications, geographic distribution and experimental parameters as well as the reasoning behind the protocols chosen for comparison. In Section 6.2, we present and reason about the results of our experiments.

## 6.1 Experimental Setup

For our WAN experiments we deployed Alea-BFT, Dumbo1/2 and HoneyBadgerBFT on $4$, $8$, $16$, $32$, $64$ and $128$ Amazon EC2 t2.medium instances, uniformly distributed across $10$ different regions (Paris, London, Frankfurt, Singapore, Tokyo, Mumbai, California, Virginia, Central Canada and St. Paulo) therefore spanning $4$ continents. Each instance was equipped with $2$ virtual CPUs, $4$GB of memory and running Amazon Linux $2$. We split our experiments in test groups based on the system scale $N$ and a varying batch size $B$ ranging from $4$ up to $10^6$ transactions. We use a fixed transaction size of $250$ bytes across all our experiments.

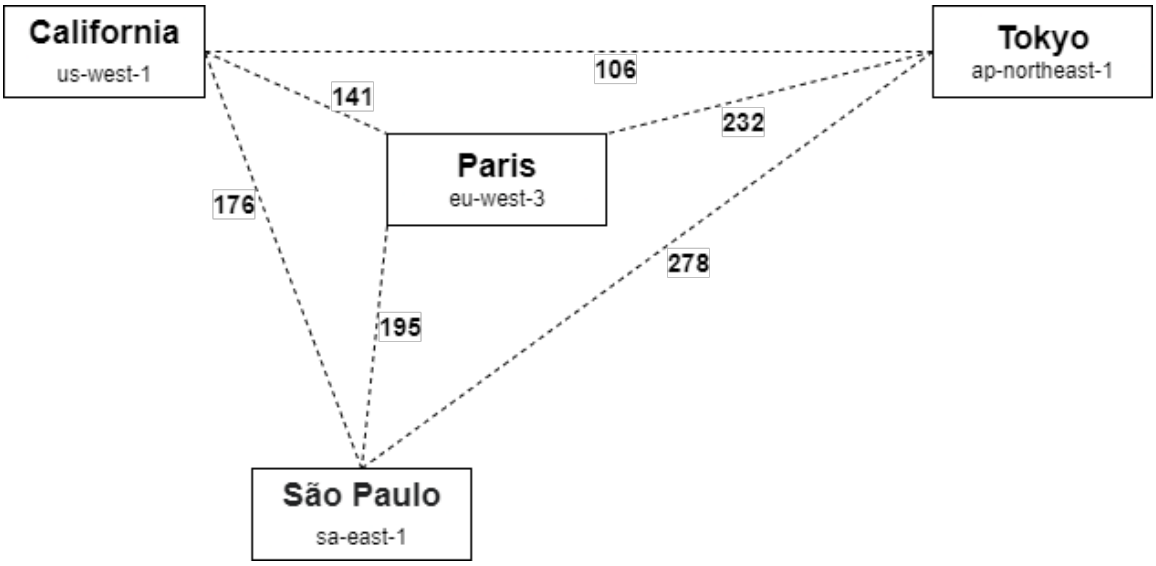



**Figure 6.1:** The experimental setup with 4 replicas, with round-trip times in milliseconds.

Note that we always attempt make instances as heterogeneously distributed as possible across continents/regions. For example, when only 4 nodes are tested, each one is located in a different continent/region, as shown in Figure 6.1.

We use HoneyBadgerBFT, commonly referred to as the first practical asynchronous broadcast protocol, as the most basic comparison baseline for our experiments. Additionally, the Dumbo protocols are included in our experiments in order to provide a comparison of Alea-BFT against the state of the art

protocols in its model, as both these protocols have been demonstrated to outperform HoneyBadgerBFT. We excluded the BEAT protocols from our experimental evaluation, as their work keeps the structure of HoneyBadgerBFT intact and most of the methods presented in it are orthogonal and compatible with the other ACS based protocols being evaluated.

## 6.2 Results

In this section we present the results of our experimental evaluation under the conditions of Section 6.1. We start by estimating the real value of $\sigma$ in an attempt to more precisely determine the analytical complexity of Alea-BFT, which will serve as support when analysing its performance metrics. Then we proceed to compare the throughput and latency of Alea-BFT, HoneyBadgerBFT and Dumbo1/2 for varying system sizes and loads. Finally, we compare the performance of these protocols under different faults scenarios.

### 6.2.1 Measuring Sigma ($\sigma$) and Message Complexity

In Section 4.5 we presented a theoretical analysis on the complexity of Alea-BFT. The analysis showed that all complexity metrics are dependent on the value of a variable $\sigma$, corresponding to average the number of ABA executions per delivered proposal, which is theoretically unbounded in the presence of a particularly adversarial network scheduler. In an attempt to quantify the actual value of $\sigma$ under realistic network conditions we measured the number of messages generated by correct processes during protocol execution for different system scales using a constant batch size of $1000$ transactions.

Figure 6.2 compares the average number of messages generated by each correct process during the execution of Alea-BFT, in order to deliver a single proposal, against a theoretical simulation for different $\sigma$ values. As we can see the experimental measurements follow very closely the theoretical simulations for which $\sigma = 1$, independently from the system scale, further supporting our hypothesis that despite being theoretically unbounded, under a realistic deployment scenario, the value of $\sigma$ does in fact converge to optimal value of $1$.

A follow-up question is how does this translate, in practice, in terms of relative message complexity when compared to the state of the art. To this end, we measured experimentally the relative message complexity per replica of Alea-BFT in comparison with HoneyBadgerBFT and Dumbo1/2. As we can see in Figure 6.3, this metric scales exponentially for the protocols based on ACS while staying linear for Alea-BFT. This is expected as in an ACS framework every replica must RBC broadcast its proposals for that batch which incurs $\mathcal{O}(N^2)$ messages per replica, while in Alea-BFT the broadcast primitive used has a message complexity of $\mathcal{O}(N)$.

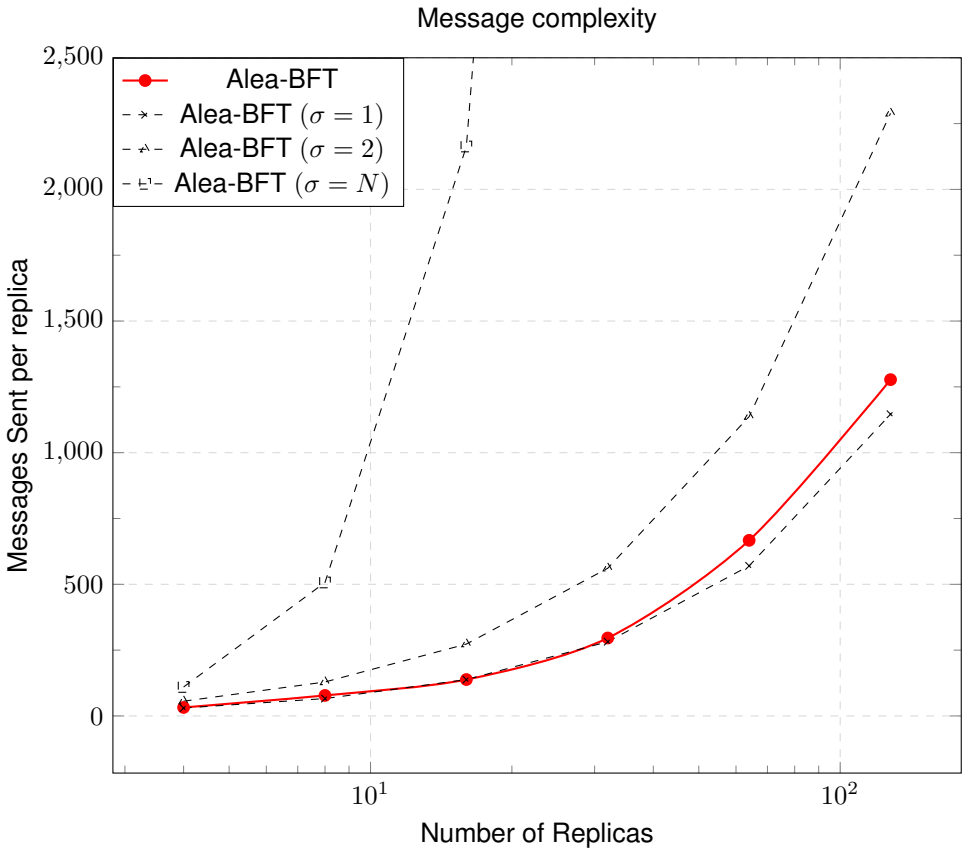



**Figure 6.2:** Messages generated per replica per batch delivered (Alea-BFT).

Additionally, to cover an unfavorable scenario where $\sigma$ increases to a high value, we also include experimental measurements for Alea-BFT using a malicious scheduler that purposely delayed the delivery of VCBC instances by $N - f$ replicas in order to artificially increasing the value of $\sigma$ to $N$. As illustrated in Figure 6.3, even under unrealistically adversarial network conditions Alea-BFT still requires fewer message exchanges than HoneyBadgerBFT, despite exceeding both Dumbo protocols. This makes sense, as for a scenario where $\sigma = N$, both Alea-BFT and HoneyBadgerBFT require $N$ ABA executions per consensus instance while Dumbo1 and 2 reduce this number to a small value $k$ (independent of $N$) and a constant value, respectively.

**Table 6.1:** Comparison of atomic broadcast protocols, assuming $\sigma$ is constant.

| **Protocol** | Message | Communication | Time |
|---|---|---|---|
| HBBFT | $\mathcal{O}(N^3)$ | $\mathcal{O}(N^2\|m\| + \lambda n^3 \log N)$ | $\mathcal{O}(\log N)$ |
| Dumbo1 | $\mathcal{O}(N^3)$ | $\mathcal{O}(N^2\|m\| + \lambda n^3 \log N)$ | $\mathcal{O}(\log k)$ |
| Dumbo2 | $\mathcal{O}(N^3)$ | $\mathcal{O}(N^2\|m\| + \lambda n^3 \log N)$ | $\mathcal{O}(1)$ |
| Alea-BFT | $\mathcal{O}(N^2)$ | $\mathcal{O}(\lambda N^2 + N^2(\|m\| + \lambda))$ | $\mathcal{O}(1)$ |

The previous experiments help corroborate our initial hypothesis regarding the practical value of $\sigma$

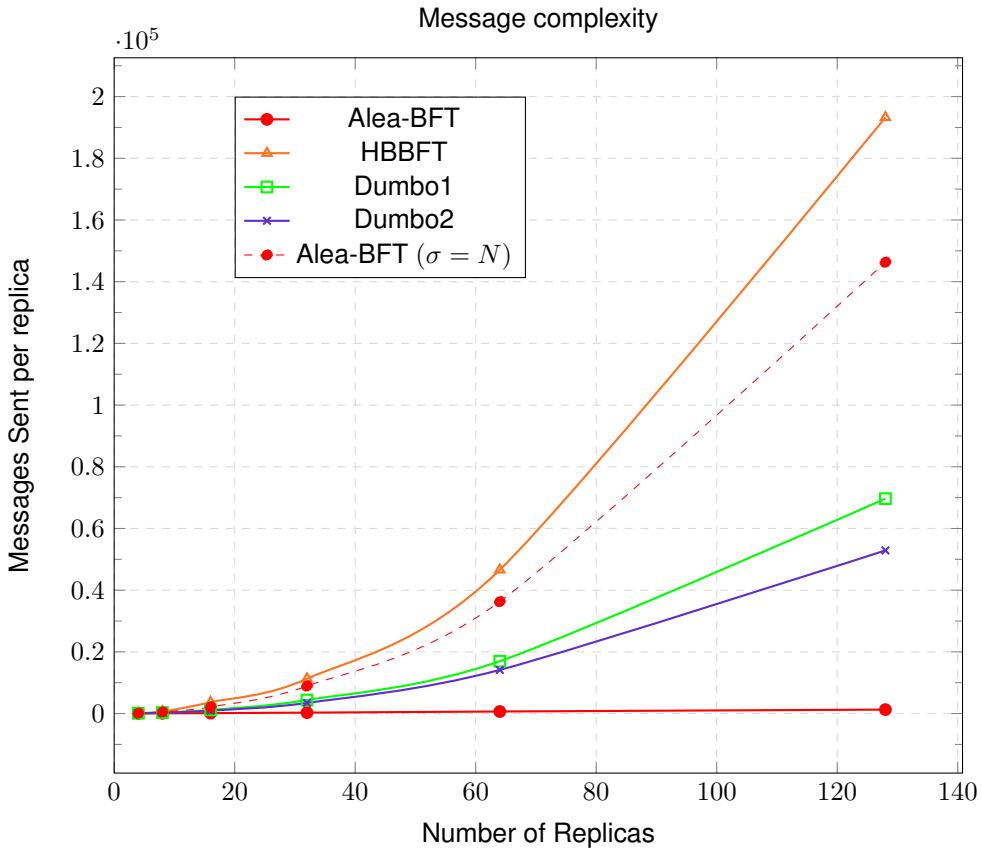



**Figure 6.3:** Messages generated per replica per batch delivered.

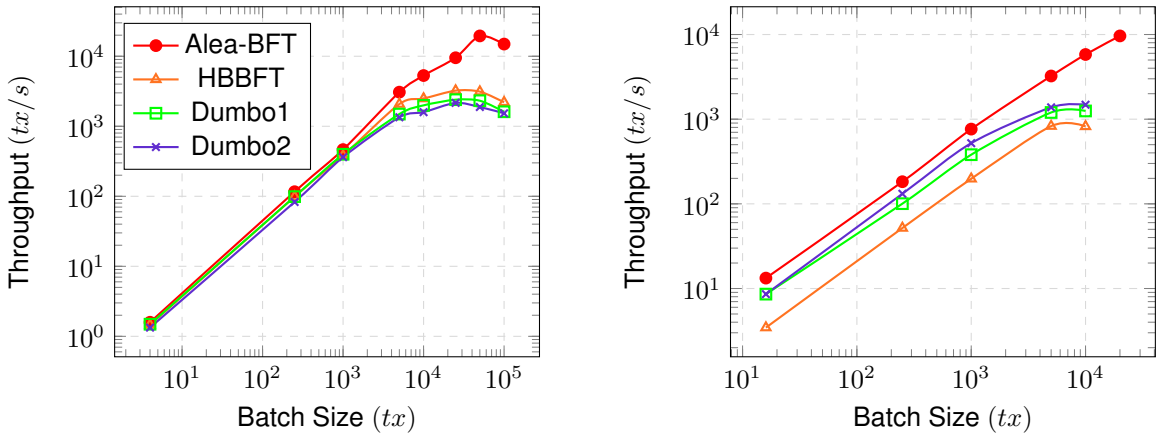
converging into a constant. Table 6.1 presents the expected complexities of Alea-BFT, HoneyBadgerBFT and Dumbo1/2 when setting $\sigma$ to the measured value of 1, further improving the state of the art, for asynchronous atomic broadcast protocols, by a factor of $\mathcal{O}(N)$ in terms of both message and communication complexity.

### 6.2.2 Throughput

Throughput is defined as the rate at which requests are serviced by the system, in case of a transaction processing system throughput is measured as the number of transactions committed by unit of time. In our experiments we measured the throughput by launching multiple replicas executing the protocol being evaluated while periodically registering the number of transactions committed during that interval.

Section 6.2.2, compares the throughput of Alea-BFT, HoneyBadgerBFT and Dumbo1/2, for different system scales, as the batch size increases. The positive slopes for the measurements pertaining to Alea-BFT indicate that our experiments did not fully saturate the available bandwidth, and it would be possible to attain higher throughput by increasing the batch size. In contrast, for HoneyBadgerBFT and

Dumbo1/2, the overall throughput of the system initially increases linearly to the increase of system load, but eventually the throughput stops increasing in some cases even start decreasing. This result follows naturally from the differences in communication complexity between the protocols, as presented in Table 6.1.




**(a)** Using 4 replicas

**(b)** Using 16 replicas

**Figure 6.4:** Throughput with varying batch sizes.

For the remaining of this section we will compare the performance of these protocols under the same batch size as we assume the available bandwidth is ample and not a bottleneck, However, readers should be aware that Alea-BFT will provide much higher throughput if using a larger batch size.

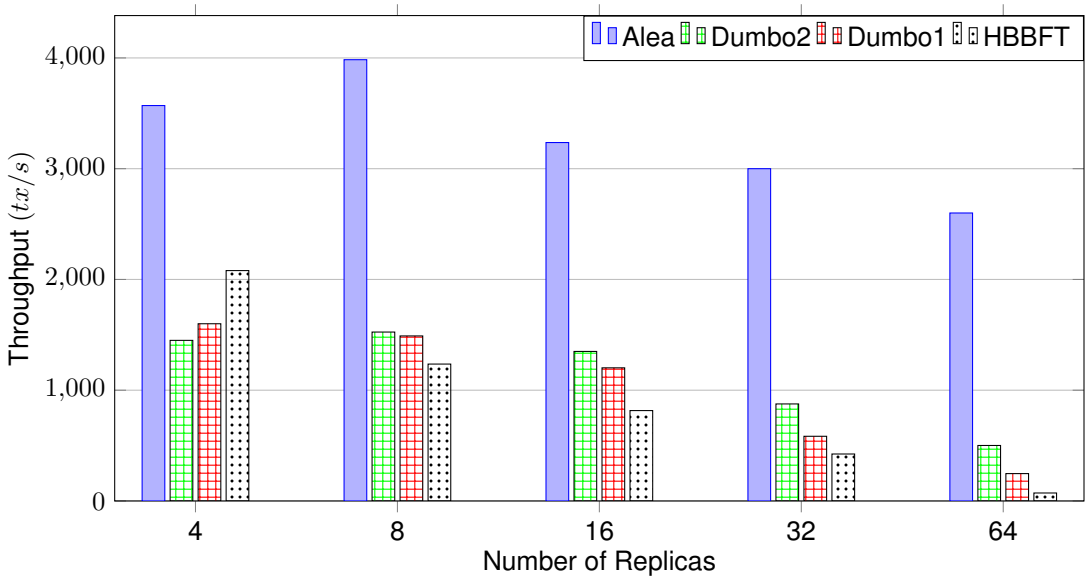



**Figure 6.5:** Throughput of Alea-BFT, HoneyBadgerBFT and Dumbo1/2 with a batch of $5x10^3$ txs.

In Figure 6.5 we present the throughput of the protocols using a fixed batch size of $5000$ transactions. We start by noticing that Alea-BFT not only outperforms all other protocols for all systems scales, but this discrepancy in performance actually increases with the number of replicas in the system revealing better scalability. This is expected since Alea-BFT presents lower message and communication complexities that its counterparts while also reducing the number of expensive threshold cryptography operations required to commit a transaction batch. Another interesting remark is that for smaller values of $N$, HoneyBadgerBFT actually outperforms both Dumbo protocols despite being theoretically more expensive.

### 6.2.3 Latency

Latency is defined as the time interval between the instant the first correct replicas start the protocol and $(N - f)$ replicas deliver the result. In our experiments we measured the latency from the instant replicas select a transaction to propose, from the pending buffers, until the instant $(N-f)$ deliver it to the application layer. In HoneyBadgerBFT and Dumbo1/2 this corresponds to an instance of ACS plus the threshold decryption round, while in Alea-BFT it encompasses the full pipeline and including the period during which a transaction is waiting in the priority queues for the agreement component to select it for delivery.

In Figure 6.6, we examine the average latency of the protocols under no contention for different system sizes, having each node propose a single transaction at the time while no other requests are being made. As we can see for small values of $N$, the basic latency of all four protocols is very similar. However, as the system size increases we start to observe some considerable difference, with the average latency of HoneyBadgerBFT increasing much faster than the remaining protocols. This discrepancy can be explained by the latency overhead associated with running multiple ABA instances, a factor that is greatly reduced in both Alea-BFT and the Dumbo protocols. Note that the basic latency of Alea-BFT is greatly influenced by the replica which proposes the transaction, as the priority queues containing ordered proposals are traversed in a round robin manner by replica id. Particularly, a proposal from a low id replica will result in a much smaller basic latency measurement than if the proposal originated from a replica with an higher identifier. For this particular experiment the proposer was randomly selected and the results averaged across multiple runs.

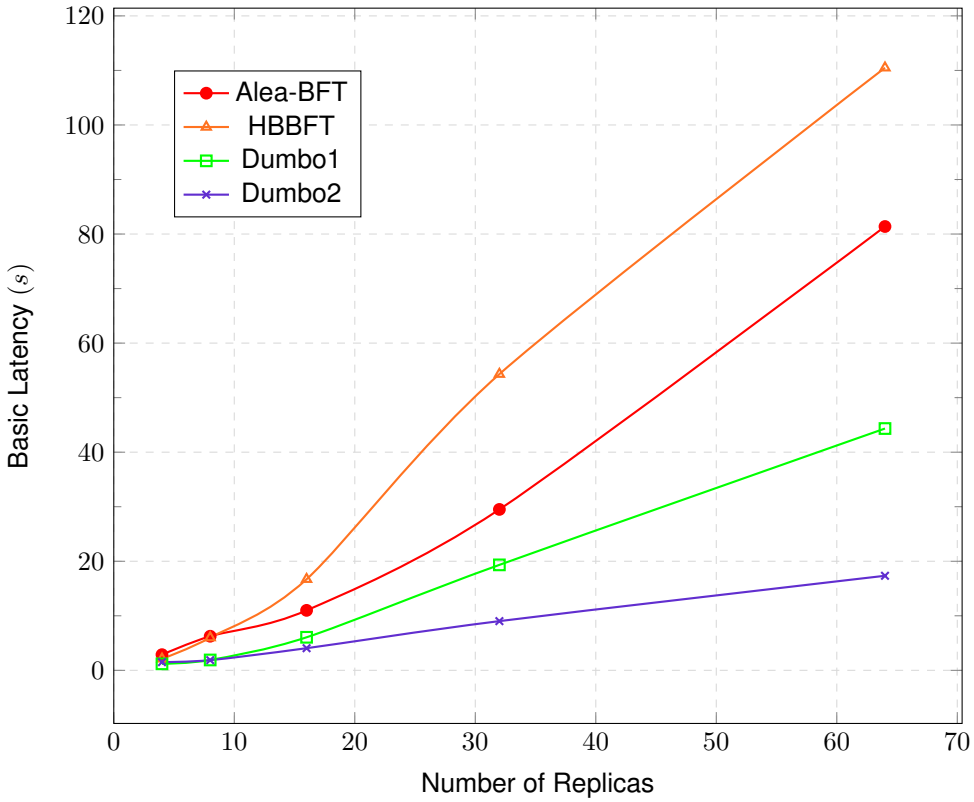



**Figure 6.6:** Basic latency.

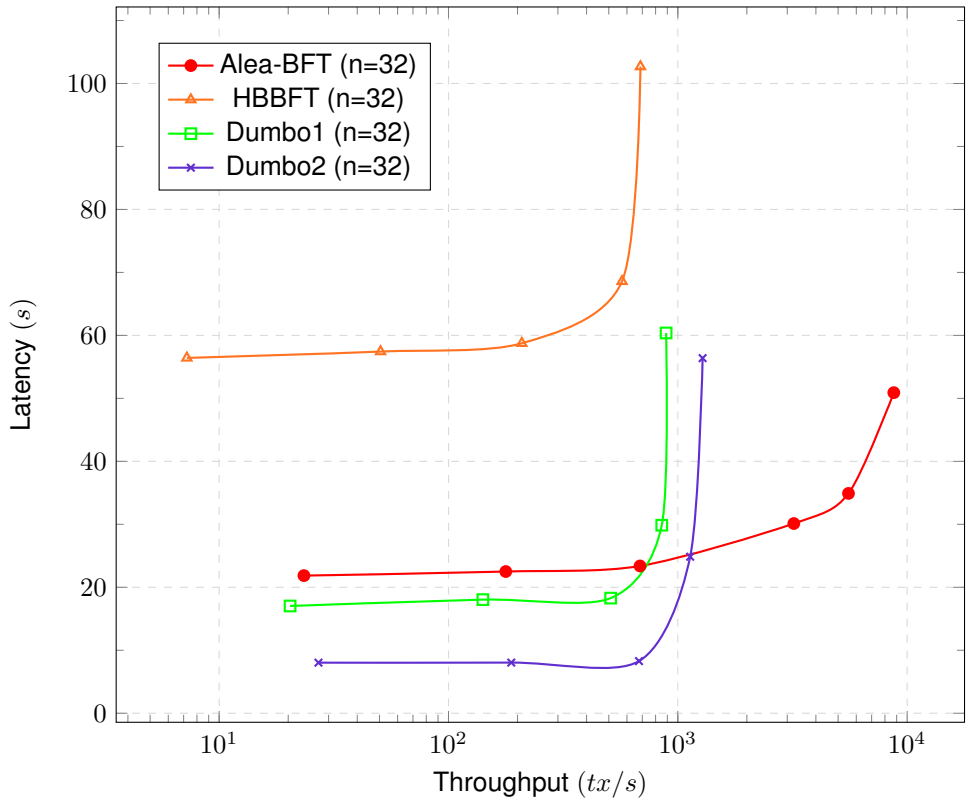





**Figure 6.7:** Throughput vs Latency.

Figure 6.7 shows how latency evolves as the system load increases for a medium system scale of 32 replicas. For all protocols, initially the latency stays relatively stable only presenting small increases as the system load grows. However, as we reach the nominal capacity of each protocol we start to notice a very steep increases in latency as the system resources stop being able to keep up with the increase in system load. We notice that Alea-BFT is able to sustain a stable latency for much higher system loads than all the other protocols due to its higher throughput and optimized bandwidth usage.

### 6.2.4 Performance Under Faults

We now evaluate the performance of Alea-BFT under faults. In Figure 6.8, we compare the performance of Alea-BFT and HoneyBadgerBFT in three different fault scenarios: failure-free, crash failure and Byzantine. In the crash failure scenario we modify the benchmark application of Section 5.3, such that $f$ replicas completely ignore all external events. In the Byzantine fault scenario we simulate the attack described in Section 4.1, in which an attacker purposely submits invalid proposals to be ordered that despite consuming resources do not count for the overall throughput of the protocol. For all experiments the system scale was set to 4 replicas, with $f = 1$, and the batch size to $10^3$ transactions.

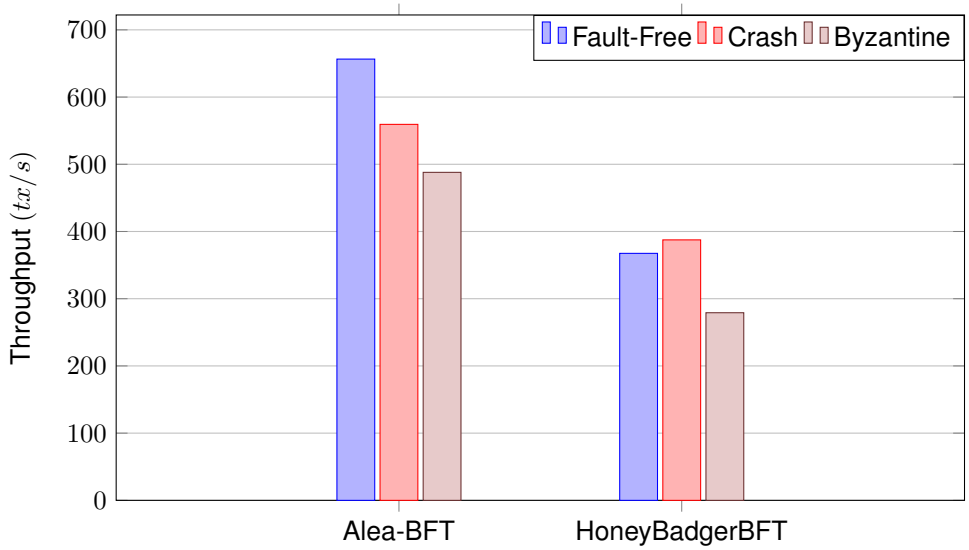



**Figure 6.8:** Throughput of Alea-BFT and HoneyBadgerBFT for different fault scenarios.

For the crash fault scenario, we obtain conflicting results. On one hand the throughput of HoneyBadgerBFT slightly increases in the presence of crash failures, while on the other hand Alea-BFT presents a slight decrease. The increase in performance of HoneyBadgerBFT follows directly from the properties of ACS. The protocol still outputs proposals from $N - f$ replicas (the ones that didn't crash), but the overall resource consumption is lower when $f$ replicas crash. The small throughput decrease of Alea-BFT can

be explained by the fact that the protocol consumes resources attempting to decide over the queues corresponding to crashed processes, which are always empty. In practice we ”waste” an ABA every $N - f$ rounds but since no replica can have received a proposal from a faulty proposer all these binary agreement instances terminate in a single round.

For the Byzantine experiments, both protocols achieved lower throughput values than in the failure-free scenario. For instance, the performance of HoneyBadgerBFT and Alea-BFT respectively decrease by about 30% and 24%. These results are directly related with how often a Byzantine replica's proposal is delivered by the protocol. In HoneyBadgerBFT this corresponds to the threshold of proposals originating from Byzantine replicas that are included in the final output vector of ACS, which, under a fair network scheduler, follows an hypergeometric distribution over a population of size $N$ and $N - f$ draws. Contrarily, under an adversarial network scheduler the output of ACS will always contain $f$ Byzantine proposals, since it can be influenced by the delivery order of messages. In Alea-BFT the performance decay, that occurs under the fault scenarios presented above, follows directly from the queue selection function used. Particularly, the round robin criteria used in our implementation of Alea-BFT, despite being simple and fair, results in faulty replicas being selected very often. A possible improvement over our baseline implementation could explore the impact of using different queue selection functions. Additionally, for the Byzantine fault scenario presented it would be possible for Alea-BFT to ignore invalid proposals during the broadcast phase. Contrarily, ACS based protocols cannot follow this approach since the commit of which proposals to include happens before the threshold decryption round revealing its contents.

The experimental evaluation of Alea-BFT demonstrated that it is able to achieve much higher throughput values than existing asynchronous atomic broadcast protocols at the cost of increased latency when the system load is low. Additionally, in spite of providing lower minimum latency values, as the system load increases, the latency of HoneyBadgerBFT and Dumbo1/2 rapidly explodes surpassing Alea-BFT. When subject to Byzantine performance failures, Alea-BFT, presented a performance decay proportional to the number of faulty replicas in the system. This follows directly from the round robin selection of pending proposals to process and motivates future work to design more clever queue selection functions.

# 7

# Conclusion

## Contents

In this chapter we summarize the main contributions of this thesis and present some ideas for future improvements and contributions in the scope of our research.

## 7.1 Conclusion

In this thesis, we were motivated by the fact that existing BFT protocols based on timing assumptions, present a series of characteristics that make them vulnerable to performance degradation attacks, and explored the use of randomization as possible a robustness mechanism. The main contribution consists on a different method for constructing asynchronous ABC protocols, which completely sidesteps from the ACS based model popularized by HoneyBadgerBFT where most recent work in the area of asynchronous BFT has been focused. Our protocol provides significant asymptotic and practical improvements over the state of the art protocols in this model. Particularly, an expected improvement by a factor of up to $\mathcal{O}(N)$ in terms of both message and communication complexities. The experimental evaluation of Alea-BFT concluded that it is able achieve multi-fold improvements over the current state of the art protocols, specially as the system size grows, therefore also providing better scalability guarantees. Overall, the main conclusion to retain from this thesis is that randomized BFT protocols can in fact be used to deploy high performance systems, while still providing all the resilience guarantees associated with a fully asynchronous operation model.

## 7.2 Future Work

Our evaluation of Alea-BFT revealed very significant improvements over its asynchronous counterparts, however it still falls short in comparison to modern BFT protocols that rely on timing assumptions, such as HotStuff [44]. Nonetheless, Alea-BFT could integrated into a practical optimistic asynchronous atomic broadcast framework, such as the one of Bolt-Dumbo [61], which executes a faster deterministic protocol in the optimistic case but falls back into an asynchronous protocol once the fast path is unable to progress. We remark that we haven't provided any optimizations and all evaluations were performed over a baseline protocol. Some possible improvements include applying different queue selection rules our including some of the general techniques presented in BEAT [49]. Finally, it would be a valuable contribution to integrate Alea-BFT into an existing blockchain system that supports modular consensus, providing a practical alternative capable of improving resilience under adversarial network conditions.

# Bibliography


[1] B. Guo, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Dumbo: Faster asynchronous bft protocols," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 803–818.

[2] P. Aublin, S. B. Mokhtar, and V. Quéma, "Rbft: Redundant byzantine fault tolerance," in *2013 IEEE 33rd International Conference on Distributed Computing Systems*. IEEE, 2013, pp. 297–306.

[3] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," https://bitcoin.org/bitcoin.pdf, 2008, accessed: 2021-01-08.

[4] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, p. 382–401, Jul. 1982.

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

[6] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, ser. OSDI '99. USA: USENIX Association, 1999, p. 173–186.

[7] A. Singh, T. Das, P. Maniatis, P. Druschel, and T. Roscoe, "Bft protocols under fire." in *NSDI*, vol. 8, 2008, pp. 189–204.

[8] Y. Amir, B. Coan, J. Kirsch, and J. Lane, "Prime: Byzantine replication under attack," *IEEE Transactions on Dependable and Secure Computing*, pp. 564–577, 2011.

[9] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making byzantine fault tolerant systems tolerate byzantine faults." in *NSDI*, vol. 9, 2009, pp. 153–168.

[10] J. Aspnes, "Randomized protocols for asynchronous consensus," *Distributed Computing*, no. 2–3, pp. 165–175, Sep. 2003.

[11] M. Ben-Or, “Another advantage of free choice: Completely asynchronous agreement protocols,” in *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’83. New York, NY, USA: Association for Computing Machinery, 1983, p. 27–30.

[12] M. O. Rabin, “Randomized byzantine generals,” in *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, 1983, pp. 403–409.

[13] H. Moniz, N. F. Neves, M. Correia, and P. Verissimo, “Ritas: Services for randomized intrusion tolerance,” *IEEE Transactions on Dependable and Secure Computing*, pp. 122–136, 2011.

[14] J. R. Douceur, “The sybil attack,” in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, ser. IPTPS ’01. Berlin, Heidelberg: Springer-Verlag, 2002, p. 251–260.

[15] “Quorum,” 2018, accessed: 2021-01-08. [Online]. Available: https://consensys.net/quorum/

[16] Libra Association, “Libra whitepaper,” 2020, accessed: 2021-01-08. [Online]. Available: https://wp.diem.com/en-US/wp-content/uploads/sites/23/2020/04/Libra_WhitePaperV2_April2020.pdf

[17] A. N. Bessani, E. A. P. Alchieri, J. Sousa, A. Oliveira, and F. Pedone, “From byzantine replication to blockchain: Consensus is only the beginning,” *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 424–436, 2020.

[18] R. Pass and E. Shi, “Hybrid consensus: Efficient consensus in the permissionless model,” *Cryptology ePrint Archive*, 2016.

[19] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Comput. Surv.*, p. 299–319, Dec. 1990.

[20] T. D. Chandra and S. Toueg, “Unreliable failure detectors for reliable distributed systems,” *J. ACM*, vol. 43, no. 2, p. 225–267, Mar. 1996. [Online]. Available: https://doi.org/10.1145/226643.226647

[21] V. Hadzilacos and S. Toueg, “A modular approach to fault-tolerant broadcasts and related problems,” Cornell University, Tech. Rep., 1994.

[22] B. Alpern and F. B. Schneider, “Defining liveness,” *Information Processing Letters*, vol. 21, no. 4, pp. 181–185, 1985. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0020019085900560

[23] D. Dolev, C. Dwork, and L. Stockmeyer, “On the minimal synchronism needed for distributed consensus,” *J. ACM*, p. 77–97, Jan. 1987.

[24] C. Dwork, N. Lynch, and L. Stockmeyer, “Consensus in the presence of partial synchrony,” *J. ACM*, p. 288–323, Apr. 1988.

[25] T. D. Chandra, V. Hadzilacos, and S. Toueg, “The weakest failure detector for solving consensus,” *Journal of the ACM (JACM)*, vol. 43, no. 4, pp. 685–722, 1996.

[26] M. K. Aguilera and S. Toueg, “Failure detection and randomization: A hybrid approach to solve consensus,” *SIAM Journal on Computing*, vol. 28, no. 3, pp. 890–903, 1998.

[27] M. Pease, R. Shostak, and L. Lamport, “Reaching agreement in the presence of faults,” *J. ACM*, p. 228–234, Apr. 1980.

[28] D. Dolev and H. R. Strong, “Polynomial algorithms for multiple processor agreement,” ser. STOC ’82. New York, NY, USA: Association for Computing Machinery, 1982, p. 401–407.

[29] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, “Hq replication: A hybrid quorum protocol for byzantine fault tolerance,” in *Proceedings of the Seventh Symposium on Operating Systems Design and Implementations (OSDI)*, Seattle, Washington, Nov. 2006.

[30] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, “Zyzzyva: Speculative byzantine fault tolerance,” *ACM Trans. Comput. Syst.*, Jan. 2010.

[31] D. Porto, J. a. Leitão, C. Li, A. Clement, A. Kate, F. Junqueira, and R. Rodrigues, “Visigoth fault tolerance,” in *Proceedings of the Tenth European Conference on Computer Systems*, ser. EuroSys ’15. New York, NY, USA: Association for Computing Machinery, 2015.

[32] P.-L. Aublin, R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, “The next 700 bft protocols,” *ACM Trans. Comput. Syst.*, vol. 32, no. 4, Jan. 2015. [Online]. Available: https://doi.org/10.1145/2658994

[33] G. Bracha and S. Toueg, “Asynchronous consensus and broadcast protocols,” *J. ACM*, p. 824–840, Oct. 1985.

[34] R. Canetti and T. Rabin, “Fast asynchronous byzantine agreement with optimal resilience,” in *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, ser. STOC ’93. New York, NY, USA: Association for Computing Machinery, 1993, p. 42–51.

[35] G. Bracha, “Asynchronous byzantine agreement protocols,” *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

[36] A. Shamir, “How to share a secret,” *Commun. ACM*, p. 612–613, Nov. 1979.

[37] C. Cachin, K. Kursawe, and V. Shoup, “Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography,” *Journal of Cryptology*, vol. 18, no. 3, pp. 219–246, 2005.

[38] A. Mostéfaoui, M. Hamouma, and M. Raynal, “Signature-free asynchronous byzantine consensus with t ¡ n/3 and o(n2) messages,” in *PODC ’14*, 2014.

[39] M. Ben-Or, “Fast asynchronous byzantine agreement (extended abstract),” in *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC ’85. New York, NY, USA: Association for Computing Machinery, 1985, p. 149–151.

[40] C. Cachin and J. Poritz, “Secure intrusion-tolerant replication on the internet,” *Proceedings International Conference on Dependable Systems and Networks*, pp. 167–176, 2002.

[41] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, “Hyperledger fabric: A distributed operating system for permissioned blockchains,” in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys ’18. Association for Computing Machinery, 2018.

[42] R. Brown, J. Carlyle, I. Grigg, and M. Hearn, “Corda: An introduction,” 09 2016.

[43] E. Buchman, J. Kwon, and Z. Milosevic, “The latest gossip on BFT consensus,” *CoRR*, 2018.

[44] M. Yin, D. Malkhi, M. K. Reiter, G. G. Gueta, and I. Abraham, “Hotstuff: Bft consensus with linearity and responsiveness,” in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019, pp. 347–356.

[45] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung, “Spin one’s wheels? byzantine fault tolerance with a spinning primary,” in *2009 28th IEEE International Symposium on Reliable Distributed Systems*, 2009, pp. 135–144.

[46] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, “Fault-scalable byzantine fault-tolerant services,” in *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, ser. SOSP ’05. New York, NY, USA: Association for Computing Machinery, 2005, p. 59–74.

[47] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The honey badger of bft protocols,” ser. CCS ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 31–42.

[48] M. Ben-Or and R. El-Yaniv, “Resilient-optimal interactive consistency in constant time,” *Distrib. Comput.*, vol. 16, no. 4, p. 249–262, Dec. 2003. [Online]. Available: https://doi.org/10.1007/s00446-002-0083-3

[49] S. Duan, M. K. Reiter, and H. Zhang, “Beat: Asynchronous bft made practical,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2028–2041.

[50] C. Liu, S. Duan, and H. Zhang, “Epic: Efficient asynchronous bft with adaptive security,” in *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2020, pp. 437–451.

[51] C. Cachin and S. Tessaro, “Asynchronous verifiable information dispersal,” in *24th IEEE Symposium on Reliable Distributed Systems (SRDS’05)*, 2005, pp. 191–201.

[52] A. Miller. (2018) Bug in aba protocol’s use of common coin. [Online]. Available: https://github.com/amiller/HoneyBadgerBFT/issues/59

[53] G. Bracha, “Asynchronous byzantine agreement protocols,” *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/089054018790054X

[54] B. Libert, M. Joye, and M. Yung, “Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares,” *Theoretical Computer Science*, vol. 645, pp. 1–24, 09 2016.

[55] M. Ben-Or and R. El-Yaniv, “Resilient-optimal interactive consistency in constant time,” *Distributed Computing*, vol. 16, no. 4, pp. 249–262, 2003.

[56] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, “Secure and efficient asynchronous broadcast protocols,” in *Annual International Cryptology Conference*. Springer, 2001, pp. 524–541.

[57] A. Boldyreva, “Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme,” in *International Workshop on Public Key Cryptography*. Springer, 2003, pp. 31–46.

[58] M. K. Reiter, “Secure agreement protocols: Reliable and atomic group multicast in rampart,” in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 1994, pp. 68–80.

[59] E. MacBrough, “Cobalt: Bft governance in open networks,” *arXiv preprint arXiv:1802.07240*, 2018.

[60] L. Lamport, “How to write a 21 st century proof,” *Journal of fixed point theory and applications*, vol. 11, no. 1, pp. 43–63, 2012.

[61] Y. Lu, Z. Lu, and Q. Tang, “Bolt-dumbo transformer: Asynchronous consensus as fast as pipelined bft,” *arXiv preprint arXiv:2103.09425*, 2021.