Alea-BFT: Practical Asynchronous Byzantine Fault Tolerance

Diogo S. Antunes, Afonso N. Oliveira, André Breda, Matheus Guilherme Franco, Henrique Moniz, and Rodrigo Rodrigues, *Instituto Superior Técnico* (*ULisboa*) *and INESC-ID*

https://www.usenix.org/conference/nsdi24/presentation/antunes

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

Alea-BFT: Practical Asynchronous Byzantine Fault Tolerance

Diogo S. Antunes, Afonso N. Oliveira, André Breda,
Matheus Guilherme Franco, Henrique Moniz, Rodrigo Rodrigues[*]
Instituto Superior Técnico (ULisboa) and INESC-ID

Abstract

Traditional Byzantine Fault Tolerance (BFT) state machine replication protocols assume a partial synchrony model, leading to a design where a leader replica drives the protocol and is replaced after a timeout. Recently, we witnessed a surge of asynchronous BFT protocols, which use randomization to remove the need for bounds on message delivery times, making them more resilient to adverse network conditions. However, existing research proposals still fall short of gaining practical adoption, plausibly because they are not able to combine good performance with a simple design that can be readily understood and adopted. In this paper, we present Alea-BFT, a simple and highly efficient asynchronous BFT protocol, which is gaining practical adoption, namely in Ethereum distributed validators. Alea-BFT brings the key design insight from classical protocols of concentrating part of the work on a single designated replica and incorporates this principle in a simple two-stage pipelined design, with an efficient broadcast led by the designated replica, followed by an inexpensive binary agreement. The evaluation of our research prototype implementation and two real-world integrations in cryptocurrency ecosystems shows excellent performance, improving on the fastest protocol (Dumbo-NG) in terms of latency and displaying good performance under faults.

1 Introduction

The history of Byzantine fault tolerant (BFT) replication has gone through different stages throughout the years, from the initial exploration of the topic in the 1980s [34] to the start of a series of practical protocols that achieve good performance in the late 1990s [16], and more recently the real-world adoption of this class of protocols in the context of cryptocurrencies and blockchains [55].

BFT protocols must carefully navigate the constraints of the FLP impossibility result [23]. This result states that no deterministic algorithm can guarantee consensus (or, equivalently, agreement on the outcome of a client request within a replicated state machine) in a fully asynchronous system where even a single process might experience a crash failure. For many decades, the almost universally accepted way to circumvent this hurdle was by assuming a partial synchrony model, where the network is assumed to be initially asynchronous but, after an unknown point in time, delivers and processes messages within a certain time bound [21]. This model leads to a class of protocol designs where a leader can drive the execution of the protocol. In this case, after a timeout indicating that the protocol is not making progress, all replicas must cooperate in picking a new leader.

Recently, researchers picked up a different line of research that had been somewhat dormant for many years: asynchronous BFT protocols [9]. These protocols are safe and live irrespectively of any timing assumptions being met, but at the cost of probabilistic guarantees, i.e., they are provided with very high probability. Removing these timing assumptions improves protocol resilience against replica and network delays, which may be due to reasons ranging from network problems to malicious activity [17, 50]. The recent surge of interest in asynchronous BFT came after the publication of a protocol called HoneyBadgerBFT (HBBFT) [40]. Since its publication, several other protocols appeared [20,25–27,29,35,36,54], making tremendous progress in the properties of these protocols, namely their performance and asymptotic complexity.

However, while these proposals succeeded in showing that asynchronous BFT algorithms can perform well, they have yet to gain practical adoption in production systems. In our view, this can largely be due to the fact that existing protocols fall short of striking a virtuous combination of good performance and simple protocol design. The academic research community often overlooks the latter, but it can be a decisive factor in practical adoption. An illustrative example of this point, from the partially synchronous arena, is the work of Istanbul BFT [41] (also known as QBFT [7]). This protocol is widely adopted by the blockchain community [6], to a large extent due to it being simple to understand and implement, and despite it being published more than two decades after PBFT [16] and its many successors.

[*]A. Oliveira is now with Three Sigma. M. Franco is now with ssv.network.

In this paper, we present Alea-BFT, the first protocol for asynchronous BFT state machine replication that brings together top-notch performance – in terms of throughput, latency, and asymptotic complexity – with a simple and elegant design and practical adoption in real-world systems.

The main insight in Alea-BFT is that it selectively brings a key design feature from classical partially synchronous protocols, namely having a per-request designated leader replica that drives the protocol execution for that request. To avoid resorting to timeouts for leader replacement, the choice of leader can constantly rotate among all replicas, as previously done in the crash [39] and Byzantine [53] models. Then, by splitting the request execution into two phases and placing on this replica the responsibility of initiating the broadcast phase to disseminate client requests, Alea-BFT avoids redundant instances of expensive building blocks present in existing asynchronous protocols and also avoids the use of threshold cryptography to encrypt proposals replicated across processes. However, this also introduces challenges, namely that there is no guarantee that the broadcast by the leader will reach a sufficient number of replicas in time for the subsequent agreement phase. We address this challenge by including an agreement phase, pipelined with the broadcast phase, whose goal is to allow replicas to agree on whether it is safe to execute the client request. The execution can proceed if sufficient replicas received the request to reconstruct it. Otherwise, the request is locally stored in one of the queues of pending requests. This leads to a design featuring a novel combination of existing building blocks, namely using VCBC as a broadcast primitive and ABA as the driver for agreement, which are judiciously joined together to provide a simple and performant protocol.

We report on three implementations of Alea-BFT: a research prototype and two real-world implementations, one of them in the context of the SSV Ethereum distributed validator (the key technology behind staking pools), which is currently being considered to replace QBFT as its main consensus protocol in the near future [51], and another in the context of an experimental consensus layer for the subnets of Filecoin [45]. More recently, a second Ethereum distributed validator incorporated Alea-BFT in its protocol roadmap [32].

Our experimental evaluation of these three prototypes shows that Alea-BFT has excellent performance, namely with comparable throughput and better latency than the fastest available asynchronous BFT from the recent literature [24]. This combination of excellent performance, protocol elegance, and real-world adoption makes Alea-BFT a practical solution for asynchronous BFT.

The remainder of the paper is organized as follows. Section 2 surveys related work. Section 3 describes the system model and building blocks. Section 4 presents the design of Alea-BFT, and we optimize it in Section 5. Section 6 analyses its asymptotic complexity. Section 7 sketches a correctness proof. Section 8 describes our various implementations, which are evaluated in Section 9. We conclude in Section 10.

2 Related Work

The Byzantine consensus problem was formulated by Lamport et al. [34], and, over time, accumulated a large body of research in the area [4, 16, 30, 31, 38, 48]. BFT recently gained adoption in cryptocurrencies and blockchains, with several new protocols for those deployments [41, 55].

From these, the protocols that implement a form of consensus – namely state machine replication protocols [49] – face the FLP impossibility of consensus in asynchronous systems [23]. To circumvent this result, most BFT systems rely on timing assumptions such as partial synchrony [21] for liveness. This is the case, for instance, of systems such as PBFT [16] and also more recent proposals such as HotStuff [55], Kauri [44] or ISS [52]. Partially synchronous protocols can, however, be sensitive to conditions like a primary that deliberately slows down the system [17] or situations where replicas are correct but the network is unreliable [50].

As an alternative to assuming partial synchrony, randomized protocols circumvent FLP by guaranteeing the liveness property with high probability. The design for this class of protocols runs the main algorithm through multiple rounds until its nondeterministic nature allows the probability of not having liveness to be irrelevant. These protocols can then operate over a fully asynchronous model, eliminating the need for timing assumptions.

Existing asynchronous BFT protocols do not simultaneously achieve the goals of simplicity and performance, which are key for practicality. In particular, the initial asynchronous BFT protocols [10, 11, 14, 42, 47] are very elegant (sometimes described in less than 10 lines of pseudocode [11]) but have high communication costs and expected termination time. More recently, several new randomized protocols appeared. At the core of this new line of proposals is an asynchronous binary agreement (ABA) primitive, in which processes decide on the value of a single bit. These ABA protocols are then used as building blocks for atomic broadcast and state machine replication solutions. After a small set of initial proposals, namely HoneyBadgerBFT (HBBFT) [40], BEAT [20], EPIC [35], and Dumbo [27], a large number of proposals emerged over the last few years [25, 26, 29, 36, 54]. Given the relatively large literature, we only describe in detail two of these proposals, namely the pioneering work of HBBFT [40] and a recent proposal with excellent performance named Dumbo-NG [24].

HBBFT [40] is based on the observation that atomic broadcast can be built on top of an asynchronous common subset (ACS) framework by combining it with a threshold encryption scheme. In ACS, every party proposes an input value and outputs a common vector containing the inputs of at least $N - f$ distinct parties. HBBFT constructs ACS from the composition of two phases: reliable broadcast (RBC) and asynchronous binary agreement (ABA). During the broadcast phase, every replica starts an RBC instance to disseminate its proposal to all other replicas. Then, in the agreement phase,

N parallel ABA instances are invoked to decide on an 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. Here, threshold encryption prevents an adversary from selectively censoring requests by selecting which proposals to include in the ACS output vector.

To our knowledge, the best performing and state-of-the-art proposal in this area (outperforming its competitors by several-fold) is Dumbo-NG [24]. This protocol decouples a continuously running broadcast phase from a sequence of multi-valued Byzantine agreement (MVBA) instances. The broadcast phase uses a custom protocol that resembles VCBC (see Section 3), whereas the MVBA phase reuses an existing protocol, whose validity predicate is fine-tuned to check for valid threshold signatures and other protocol-specific conditions. The presence of an MVBA protocol introduces an $O(n^3)$ message complexity, which contrasts with Alea-BFT's use of a round-robin ABA, with only $O(n^2)$ complexity.

Generally, we can categorize previous proposals as either suffering from high communication costs (pre-HBBFT protocols) or having a more complex design that hinders practical adoption (new generation, starting from HBBFT). In contrast, Alea-BFT brings together a simple and elegant design with excellent performance and is now being adopted in real-world systems, namely Ethereum distributed validators. This might be in part due to the simplicity of the protocol and its components – for instance, while Dumbo-NG uses an MVBA, which is complex in both the provided guarantees and its implementation, Alea-BFT leverages a much simpler ABA primitive, resulting in an overall protocol that is easier to understand and implement. Furthermore, Alea-BFT improves on most prior asynchronous protocols through its near quadratic message complexity. Note that while quadratic protocols have been theoretically proposed [5], we do not know of any protocol with such characteristics that was implemented.

3 Basics

In this section, we present the system model and precisely define the basic blocks upon which Alea-BFT is built.

3.1 System model

We consider a distributed system composed of N processes, also called replicas, uniquely identified from the set $S = \{P_0, ..., P_{N-1}\}$ and an arbitrary number of clients.

We assume a Byzantine failure model where up to $f = \lfloor \frac{N-1}{3} \rfloor$ replica processes can fail arbitrarily during the execution of the protocol. The remaining processes follow the protocol specification and are termed correct. Alea-BFT is adaptively secure against an adversary that dynamically determines the replicas to compromise. That said, it reuses two classes of protocols described later in this section, which can have either statically secure or adaptively secure instantiations. As such, choosing a statically secure subprotocol would downgrade the solution to be statically secure.

The system is asynchronous, with the message delivery schedule under adversarial control, and without bounds on communication delays or processing times. Processes are fully connected by channels, providing guarantees that messages are not modified in transit and are eventually delivered. In practice, this requires message retransmission and point-to-point authentication, but by considering this network model, we can omit these from the protocol description.

Lastly, the adversary is assumed to be computationally bound and thus unable to subvert cryptographic primitives.

3.2 Specification

We specify Alea-BFT as an atomic broadcast protocol, which is a common abstraction for implementing state machine replication. Intuitively, this allows a process (e.g., a proxy replica) to broadcast a message (e.g., a client request[1] to be executed on the state machine) to all processes, ensuring that all processes deliver all messages in the same order (executing all client requests in the same order and therefore transitioning through the same sequence of states). Formally, atomic broadcast is defined as follows (with the standard assumption that messages include a per-sender id and sequence number to make them unique) [28]:

- *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 messages m and m', then both deliver m and m' in the same order.

3.3 Building blocks

Alea-BFT is designed in a modular way by reusing several subprotocols to carry out certain tasks. In this modular architecture, upper-level protocols provide inputs and receive outputs from subprotocols at the lower layers. Next, we present the precise specification of these underlying primitives.

3.3.1 Verifiable Consistent Broadcast Protocol

Verifiable consistent broadcast (VCBC) is a broadcast variant that was first proposed by Cachin et al. [14]. It can only guarantee that all correct replica processes deliver the broadcast value if the sender is correct; however, it always ensures that no two correct processes deliver conflicting messages. Additionally, it allows any party P_i that has delivered message m to inform another party P_j about the outcome of the broadcast execution, allowing it to deliver m immediately and terminate the corresponding VCBC instance. More formally, a VCBC protocol ensures the following properties [14]:

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

[1] Client requests are also referred to in the literature as state machine commands. Throughout the paper, we will use only the term *request*.

- **Integrity:** Every correct party delivers at most one message. Additionally, if the sender is correct, then it previously broadcast the message.
- **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 σ carried by M is independent of the length of m.

In Alea-BFT, we use a VCBC implementation consisting of extending an echo broadcast protocol [14] with threshold signatures to generate the proof σ. In short, the protocol consists of the distinguished sender process sending m to all processes and collecting a Byzantine quorum of $\lceil \frac{n+f+1}{2} \rceil$ signature shares in the replies, allowing the sender to combine these shares and convey the signature to all processes in the final message step. Using threshold signatures keeps the message size constant, ensuring succinctness. The message complexity of the VCBC protocol we use is $O(N)$ and its communication complexity is $O(N(|m|+\lambda))$, assuming the size of a threshold signature and share is at most λ bits.

3.3.2 Asynchronous Binary Agreement

An asynchronous binary agreement (ABA) protocol 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 a common value b from the set of proposals by correct processes. Formally, a binary agreement protocol can be defined by the following properties:

- **Agreement:** If any correct process decides b and another correct process decides 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.

Given the FLP theorem [23], no deterministic algorithm can satisfy all the previous properties in the asynchronous model of Alea-BFT. As such, we use a randomized solution with the following termination property:

- **Termination:** The probability that a correct process is undecided after r rounds approaches 0 as r approaches ∞.

This way, even though the number of rounds required to reach agreement is unbounded, the probability that the protocol does not terminate converges to zero.

We instantiate this primitive via the Cobalt ABA [37] protocol, a modified version of the proposal by Mostéfaoui et al. [43]. The protocol relies on a common source of randomness, i.e., a "common coin", 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 [15]. The protocol proceeds in rounds, each consisting of the following all-to-all message communication steps: INIT, conveying the most recent proposal (0 or 1) of each process, followed by AUX and CONF, trying to confirm the existence of strong support (i.e., a Byzantine quorum) in the previous step for a value. At the end of these exchanges, processes either decide a value (if that support was gathered) and convey it through a FINISH message or otherwise move to the next round, changing their proposal to the value of the common coin whenever both final outcomes are considered possible. This protocol provides optimal resilience, $O(N^2)$ expected message complexity, $O(\lambda N^2)$ expected communication complexity and terminates in $O(1)$ expected time.

4 Alea-BFT

This section presents Alea-BFT, starting with a broad overview, followed by a detailed description and pseudocode.

4.1 Overview

One of the central insights of Alea-BFT is to have a single replica propose a value per consensus instance, similar to what happens in leader-based protocols in the partially synchronous model, and all others agree on whether to deliver it or not. Departing from a design where all replicas try to insert each client request in the total order enables us to remove an all-to-all communication phase and only have a single ABA execution per client request (or batch of requests). This insight then leads to the following initial design.

Strawman Proposal. The first design consists of adapting the ACS construction of HBBFT but, instead of having all replicas simultaneously propose values, a single replica is selected as the proposer for each consensus round. The role of the proposer is to choose a value (or batch of values) from its buffer of pending requests to serve as a proposal and broadcast it to all replicas, using a broadcast primitive that ensures that all replicas receive the same value – if they output a value at all – a property ensured by consistent broadcast [13]. Correct replicas would then proceed to execute a *single* ABA to determine whether to deliver the proposed value for that round (if enough replicas have received it to ensure it persists despite faults or asynchrony) or not deliver anything. Additionally, the proposer is deterministically rotated upon every ABA execution to address the scenario where the proposer is faulty without introducing a fail-over sub-protocol, similar to what happens in other protocols for the partially synchronous model (both with crash [39] and Byzantine [53] faults) that incorporate leader rotation into the normal operation, such that it is constantly changing.

This strawman protocol, however, raises an immediate problem. In previous protocols based on an ACS framework, replicas are guaranteed to receive proposals from at least $N-f$ correct replicas. Therefore, they can wait until this threshold is met before deciding which values to input for the subsequent agreement stage. In contrast, 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, thus making it difficult to decide which value to input into the ABA without resorting to some timeout, which contradicts the asynchronous model.

Final design. The impossibility of waiting for some threshold to be met before deciding the value to input to the ABA stage

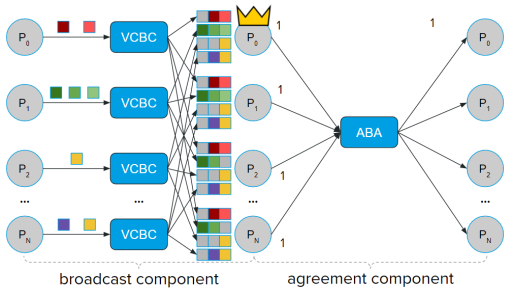


Figure 1: Overview of Alea-BFT. Requests go through a single broadcast primitive (VCBC), are inserted in a priority queue at each replica, determining the final ABA input.

leads us to the insight of not waiting at all and instead allowing undelivered proposals to exist, which are then carried over across rounds. In other words, every time a particular replica is reelected as the proposer, the corresponding ABA execution will decide over its queue of pending proposals instead of a single newly proposed value (or batch of values). This way, replicas can submit their input to start the ABA for a new round as soon as they conclude the previous round, since even if the decision is 0 (i.e., not deliver any proposal in the round), the same proposal will be eventually revisited when the same replica becomes the leader and a larger threshold of replicas become aware of the proposal, guaranteeing convergence to an ABA decision of 1 over time. The ABA execution also serves as a synchronization mechanism between replicas since no replica can progress to a round until it concludes all ABA instances for previous rounds.

In Alea-BFT, we leverage this idea to decompose the monolithic architecture of previous ACS-based protocols, 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, either by the current or by a subsequent execution of the second phase (agreement component). Very importantly for performance, these two phases are executed in parallel, allowing for efficient pipelining.

Figure 1 depicts the resulting overall protocol flow. It starts with the broadcast component of the Alea-BFT pipeline, where replicas receive client requests, (optionally) batch them by storing these in a pending buffer of size B, and, when the buffer is full, disseminate its contents via a VCBC primitive tagged with an incremental sequence number s. The output of VCBC at each replica is stored in a buffer and only removed upon a decision of 1 in the subsequent phase. The broadcast stage produces an instance of an ordered queue of undelivered proposals at each replica. These instances are then used as input to the next component of the pipeline. Note that every replica maintains N queues of undelivered proposals, one for each replica in the system, and these grow and shrink over time depending on how efficiently the agreement component can process them.

The next stage is the agreement component, which iteratively selects one of the queues and decides whether to deliver the oldest proposal. To this end, replicas participate in a single ABA execution, voting 1 if their queue contains this proposal or 0 otherwise. If the decision is 1, then a sufficient threshold of correct replicas are aware of the proposal and may safely deliver it, as explained next. Otherwise, if a decision is 0, the agreement component simply moves on to the next queue, repeating the same process.

As mentioned, since the broadcast's VCBC primitive may terminate at different times in different processes, we need to address the scenario where a correct process outputs an ABA decision of 1, but does not yet know the corresponding proposal. In this scenario, such a correct process requests the missing proposal from the other processes that voted 1. This recovery mechanism is guaranteed to work for the following reasons. Since ABA decided 1, at least one correct process voted for 1. Therefore, this process has the required VCBC proof (as guaranteed by VCBC's verifiability property) and can forward it to the requesting process.

4.2 Detailed description

Processes in Alea-BFT maintain two state variables shared between the two components of the pipeline: variable S_i, consisting of the set of all messages delivered by the protocol, which is initialized as empty upon a call to the START procedure, and updated during the execution of the agreement component; and variable $\texttt{queues}_i$, comprising an array of N priority queues, each corresponding to a distinct replica $P_x, \forall x \in \{0,...,N-1\}$. Algorithm 1 is responsible for initializing the shared state variables and starting the pipeline components upon a call to the START procedure. In the remainder of this section, we begin by specifying the data structure of the priority queues and then describe the two components of the pipeline in turn.

Algorithm 1 Alea-BFT - Initialization (at P_i)

```
1:  constants:
2:      N
3:      f
4:  state variables:
5:      S_i ← ∅
6:      queues_i ← ∅
7:  procedure START
8:      queues_i[x] ← new pQueue() , ∀x ∈ {0,...,N−1}
9:      async BC-START()
10:     async AC-START()
```

4.2.1 Priority queues

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, where the lower-numbered priority values represent the elements that must be

processed first. Only a single element can be inserted in a given slot, even after being removed, as the slot is permanently labelled as used and cannot store another element. A special slot called the head slot always points to the slot with the lowest-numbered priority whose value has not been removed yet. The pointer to the head slot progresses incrementally, conditioned by the removal of elements from the queue. A priority queue exposes the following attributes:

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

Additionally, a priority queue provides an interface for accessing and modifying its contents as described below:

- **Enqueue** (v,s)**:** Add an element v with a given priority value s to the queue (ignored if the corresponding slot is not empty).
- **Dequeue** (v)**:** Removes all instances of the specified element v from the queue if it is present.
- **Peek** $() \rightarrow \{v, \perp\}$**:** Retrieve, without removing, the element v in the head slot of the queue or $\perp$ if the slot is still empty (because no **Enqueue** for that slot has been invoked yet).

As we will see, Alea-BFT leverages the properties of this structure to mediate the communication between the broadcast and agreement components of the protocol pipeline. In particular, each of the N priority queues that each replica maintains keeps track of the undelivered proposals originating from the other replicas, ordered by the priority value assigned to those proposals.

4.2.2 Broadcast Component

The broadcast component is responsible for establishing an initial local order over the client updates received and propagating that order to other replicas. Every replica process maintains two local state variables, a buffer of pending client requests buf_i, and an integer value $priority_i$, indicating the next sequence number it should assign to a proposal. The main logic of this component, illustrated in Algorithm 2, is split between two upon rules:

Upon rule 1 (lines 9 to 15): The first rule is triggered at process P_i upon receiving a client message m to be broadcast in total order. It is responsible for waiting until a batch of B requests has been accumulated, assigning it a local sequence number, and VCBC-broadcasting it to all replicas. In more detail, process P_i 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 buf_i reached a threshold B, input buf_i to a VCBC instance tagged with ID $(i, priority_i)$, indicating that P_i assigned the local priority value $priority_i$ to a proposal consisting of the current buffer contents (lines 12 to 13).
- Increment $priority_i$, so that it can be assigned to the next proposal from P_i, and clear the buffer (lines 14 to 15).

Upon rule 2 (lines 16 to 20): The second rule is triggered at 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 assigned to it by P_j. Process P_i proceeds as follows:

- Insert the delivered proposal m into the slot $priority_j$ of the priority queue Q_j, mapping to P_j (lines 17 to 18). This corresponds to P_i updating its view on the state of P_j's pending requests.
- If the set S_i contains m, indicating that it had already been delivered, then process P_i immediately removes it from Q_j to prevent a duplicate delivery that would violate the integrity property (lines 19 to 20).

Algorithm 2 Alea-BFT - Broadcast Component (at P_i)

```
1:  constants:
2:      B
3:  state variables:
4:      buf_i
5:      priority_i
6:  procedure BC-START
7:      buf_i ← ∅
8:      priority_i ← 0
9:  upon receiving a message m, from a client do
10:     if m ∉ S_i then
11:         buf_i ← buf_i ∪ {m}
12:         if |buf_i| = B then
13:             input buf_i to VCBC (i, priority_i)
14:             buf_i ← ∅
15:             priority_i ← priority_i + 1
16: upon outputting m for VCBC (j, priority_j) do
17:     Q_j ← queues_i[j]
18:     Q_j.Enqueue(priority_j, m)
19:     if m ∈ S_i then
20:         Q_j.Dequeue(m)
```

4.2.3 Agreement Component

The agreement component presented in Algorithm 3 establishes a total order among client requests. Requests are ordered through a succession of agreement rounds that iterate through the various priority queues and decide whether to insert the head of that queue in the total order or skip it. Processes maintain a single state variable r_i, serving as a 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): For each iteration r_i of the agreement loop, the queue of proposals pertaining to a certain replica is selected. This replica is a designated round leader, chosen through a deterministic function of the round number F (e.g., by rotating through all replicas). Let P_a denote the current round leader, and Q_a the corresponding priority queue at each replica r_i. Process P_i proceeds as follows:

- Run an ABA instance with id (r_i) to determine whether the *value* in the head slot of Q_a should be delivered in this round. Process P_i, inputs 1 to ABA if its local Q_a contained *value* in the head slot, or 0 otherwise (lines 6 to 9).
- If the ABA execution decided for 0, indicating that no proposal should be delivered for the current round r_i, simply proceed to the next loop iteration, otherwise:
  - If process P_i input 0 to ABA, send a FILL-GAP message to all processes that voted for 1. This step 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 (line 14). The value of the head slot can be updated by the delivery of a pending VCBC instance, either through "normal" execution or as a result of the reception of a FILLER message.

In addition to the main agreement loop, the agreement component also defines two upon rules associated with the recovery sub-protocol 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 queue pertaining to P_q is more advanced than the one of P_j, by comparing the head pointer of its Q_q against s (line 19). If it is lower, P_i cannot satisfy the FILL-GAP request and thus ignores it. Otherwise:
  – Compute and store in *entries* a verifiable message M for all VCBC instances originating from P_q tagged with a priority comprised between the value s, requested by P_j, and the 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 receiving a valid $\langle \texttt{FILLER}, entries \rangle$ message. This message is received as a response to a FILL-GAP request. It 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 the verifiability property of VCBC ensures that it immediately terminates upon the reception of M, therefore triggering 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 the contents of *value*, a batch of totally ordered messages m, to the application layer (line 31). Additionally, this procedure also removes *value* from all priority queues and appends its contents to the set of delivered requests S. Note that if batching is naively used, this scheme would likely lead to some redundant work being done by the replicas, as large batches differing only in a few requests could not be removed from the priority queues (in line 27), and therefore redundant operations would go through agreement and only be removed before attempting to execute them (line 29). To avoid this, we steer the protocol towards all replicas having the same batches by having the client optimistically submit requests to a single replica. If, after a timeout, the client does not receive a response, then it resubmits to all replicas. Furthermore, a real-world implementation would place an upper bound on the number of broadcast but not delivered requests, which implies that requests are not batched as soon as they are received but instead stay in a pool until the protocol progresses. Because of this, deduplication can be made before the batch is created, avoiding the redundant work problem.

Algorithm 3 Alea-BFT - Agreement Component (at P_i)

```
1:  state variables:
2:      r_i
3:  procedure AC-START
4:      r_i ← 0
5:      while true do
6:          Q ← queues_i[F(r_i)]
7:          value ← Q.Peek()
8:          proposal ← value ≠ ⊥ ? 1 : 0
9:          input proposal to ABA (r_i)
10:         wait until ABA (r_i) delivers b then
11:             if b = 1 then
12:                 if Q.Peek() = ⊥ then
13:                     broadcast ⟨FILL-GAP, Q.id, Q.head⟩
14:                 wait until (value ← Q.Peek()) ≠ ⊥ then
15:                     AC-DELIVER(value)
16:         r_i ← r_i + 1
17: upon receiving a valid ⟨FILL-GAP, q, s⟩ message from P_j do
18:     Q ← queues_i[q]
19:     if Q.head ≥ s then
20:         entries ←VCBC(queue, s').M ∀_{s'} ∈ [s, Q.head]
21:         send ⟨FILLER, entries⟩ to P_j
22: upon delivering a valid ⟨FILLER, entries⟩ message do
23:     for each message M ∈ entries do
24:         deliver M to the corresponding VCBC
25: procedure AC-DELIVER(value)
26:     for each Q ∈ queues_i do
27:         Q.Dequeue(value)
28:     for each m ∈ value do
29:         if m ∉ S_i then
30:             S_i ← S_i ∪ {m}
31:             output m
```

5 Optimizations

As we implemented and tested Alea-BFT, we developed the following optimizations to improve its performance.

Input unanimity. When a replica observes all N replicas providing as input the same value v to an ABA instance, then it is guaranteed that the ABA will decide v. To leverage this observation, we added an early termination path to the ABA protocol. This is achieved by modifying the `INIT` message (which is only sent once at the start of the protocol) to convey the input of each replica. Then, when a replica receives N modified `INIT` messages with the same value v, it immediately delivers v and broadcasts `FINISH` (if not broadcast yet). Crucially, it continues executing the ABA protocol normally until it receives $2f+1$ `FINISH` messages, as only then it is guaranteed that all correct replicas can eventually terminate.

Pipelining prediction. To maximize the chances of a successful outcome of the ABA stage, replicas keep statistics about the time to complete previous VCBC and ABA executions and use that information to fine-tune the pipeline and adapt it to the network conditions. In particular, replicas delay negative votes for an ABA when a VCBC for the slot being voted is still in progress but is expected to end soon (according to the current estimate), with the expectation that the time to complete the broadcast is smaller than the cost of a negative ABA result. Additionally, replicas anticipate batch formation (and consequently the start of VCBC) when deemed useful to minimize the chance of a negative ABA result. This is achieved by attempting to time the start of the broadcast, such that it ends right before the corresponding ABA.

Leader prediction. Latency can be improved if the client sends the request to a replica that is about to become a leader: if that happens, that replica will quickly include it in the next batch to be processed and delivered. In situations with low load and where a single client issues a sequence of requests, we found that using a round-robin approach is very effective because the rate of requests followed the leader rotation. Alternatively, clients can receive periodic hints from the replicas about the rotation schedule or rely on the replica they contact to redirect the request to a faster replica.

6 Analysis

This section analyzes the asymptotic efficiency of the Alea-BFT protocol according to time, message, and communication complexity metrics. The results of this analysis are summarized in Table 1.

To analyze Alea-BFT we observe that message exchanges occur in three places for each proposal payload to be delivered. First, during the execution of the broadcast component, a replica initiates a VCBC instance to disseminate the locally ordered proposal. Second, all replicas participate in successive ABA executions to decide whether or not to deliver the proposal in a particular slot. Here, we denote by σ the average number of ABA instances executed over a given slot to reach a positive decision. Finally, a fetch request is triggered by replicas that did not VCBC-deliver the proposal before the corresponding ABA decided 1.

6.1 Time Complexity

Time complexity is defined as the expected number of communication steps from a client request to its output. In the case of Alea-BFT, the first and third steps terminate in constant time $O(1)$. In contrast, the total number of rounds required for the agreement component to decide depends on the value of σ, therefore giving an expected time complexity of Alea-BFT of $O(\sigma)$.

6.2 Message Complexity

We measure message complexity as the expected number of messages generated by correct replicas to execute a single client request. In Alea-BFT, the VCBC instance from the broadcast phase generates $O(N)$ messages; then, every ABA instance exchanges $O(N^2)$ messages in expectation; and finally, the third recovery phase incurs an overhead of $O(N)$ messages per replica that triggers this fallback protocol. Hence, the expected message complexity of Alea-BFT is $O(\sigma N^2)$, due to the σ ABA instances that are executed per priority queue slot before delivery, which is close to the quadratic lower bound on message complexity shown by Dolev and Reischuk [19].

6.3 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 λ the size of a threshold signature share. The execution of VCBC incurs a communication complexity of $O(N(|m|+\lambda))$. Each ABA instance requires correct replicas to exchange $O(\lambda N^2)$ bits in expectation, and finally, each replica that triggers the recovery phase adds communication cost of $O(N(|m|+\lambda))$ bits. This results in an expected total communication complexity of $O(N^2(|m|+\sigma\lambda))$ due to σ ABA executions and up to N recovery rounds being triggered.

Table 1: Complexity of Alea-BFT decomposed by stages.

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

6.4 Estimating σ

As previously mentioned, Alea-BFT does not guarantee a constant-time execution, which could negatively affect the protocol latency. In particular, this is because multiple zero-deciding ABA instances could be executed over the same priority queue slot until its contents are considered totally ordered. However, we argue that, despite being theoretically

unbounded, the value of σ (the number of ABA instances required for a decision) is, in practice, close to the optimal value of 1. This is justified by the observation that, in a round-robin leader assignment, each queue is revisited every N rounds, meaning that $N-1$ other ABA instances were executed by the time a given queue is revisited. Considering ABA's validity property, which states that the decided value must have been proposed by a correct process, 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, for the value of σ to increase by a single unit, correct replicas would, on average, have to complete N sequential ABA executions for every single VCBC instance. In our experiments, we validated that the value of σ was very close to 1 in practice.

7 Correctness

Next, we sketch the correctness of the protocol, and we provide a complete proof in a separate arXiv e-print [8]. The complete proof follows the structured proof format by Lamport [33] for increased preciseness.

Safety. The two safety properties that need to be proven are integrity (each message m appears at most once in the delivery sequence of correct process i) and total order (any two messages m and m' are delivered in the same order by any pair of correct processes i and j). Integrity is derived from the fact that once a message is delivered, it is added to the set of delivered messages, dequeued from all queues and never enqueued again. The total order property follows from the fact that delivering two different messages at different replicas in the same slot would lead to a violation of the consistency property of VCBC.

Liveness. The liveness properties build mostly on the liveness guarantees of the protocols used as building blocks. Given these guarantees, it suffices to follow the protocol steps to prove that we eventually satisfy the preconditions for the building block protocols to produce the necessary outputs to decide a value, namely that the messages that were broadcast reach a sufficient number of correct processes.

Censorship resilience. Prior asynchronous BFT protocols include mechanisms to enforce that Byzantine replicas cannot significantly delay the delivery of any particular message (i.e., a fairness property). This is required, in particular, for protocols that use an asynchronous common subset (ACS) to agree on a subset of the various proposals from different replicas to deliver, since a Byzantine replica can bias the choice of proposals to be included in the output of ACS. In Alea-BFT, however, censorship resilience is easily achieved by construction, given that any replica can initiate a VCBC for a client request. Thus, clients can broadcast their requests to $f+1$ or more replicas (possibly after a wait, to optimistically check if sending to a single replica suffices). This guarantees that at least one of these replicas is non-faulty and will drive the request execution.

8 Implementations

We implemented Alea-BFT in three open-source prototypes: an initial research implementation, and then two real-world integrations, namely with the SSV Ethereum distributed validator (where it is being considered to replace QBFT as its main protocol in the near future [51]) and with an experimental consensus layer for subnets in the Filecoin network [1].

Research prototype. Our first prototype implementation of Alea-BFT, which is available as open source [2], comprises 20,000 lines of Java code. The source code is organized in a modular manner, with the main logic of Alea-BFT leveraging different subprotocols (namely broadcast and binary agreement) as building blocks. Reliable point-to-point links were implemented using TCP streams, similar to prior work [24, 27, 40]. Additionally, we implemented HBBFT using the same codebase as a starting point to use it as one of the comparison baselines.

Ethereum distributed validator. Decentralized Validator Technology (DVT) is a technology to improve the security, robustness, and openness of the Ethereum network [12, 22]. Using distributed validators, several non-trusted parties cooperate to logically act as a single validator, and this way each participant is able to overcome the need to commit 32 ETH (over USD 3,700 as of this writing) to enter the network. To act as a single logical entity, once a distributed validator is called to conduct a validation task (called a duty in Ethereum), the various parties that form the validator run a BFT consensus protocol to decide on the input to the duty (which can be, for instance, a block or a pointer to the head of the chain, depending on the type of duty) and the respective outcome.

We have been collaborating with ssv.network for over one year [51] to implement Alea-BFT in the SSV codebase, with the goal of offering stronger resilience in the presence of adverse network conditions or Byzantine behavior. Their current plan is to incorporate Alea-BFT in their production codebase in the near future. The repository for this implementation is available as open source [3].

The main integration challenge came from the fact that consensus is used as a standalone instance, instead of a replicated state machine that executes a command sequence, which would be more aligned with the abstraction offered by Alea-BFT. Therefore, we adapted Alea-BFT with the following design features and optimizations to fit this specific context.

Adapting Alea-BFT to one-shot consensus. In a distributed validator, even though the duty is the same across the processes that comprise the validator, because it is known a few epochs in advance, the input to that duty needs to be agreed upon because each process may retrieve it from a different source (called a beacon client in Ethereum, with several possible providers). To reach consensus on that input, each process will attempt to send its input using Alea-BFT's atomic broadcast protocol, and the first to be delivered by the protocol is the output of consensus. Note that only one instance of VCBC per process is needed to implement this one-shot consensus, thus

simplifying the implementation. A possible concern is that the validity condition for this consensus implementation allows for a single divergent opinion to be the final output. However, this is safe because the inputs are coming from sources that are outside of the distributed validator system. Therefore their correctness is beyond the scope of the distributed validator. (In addition, some basic validation checks can also be conducted.) Additionally, for fairness, the round-robin rotation of protocol leaders in different consensus instances is based on a pseudorandom sequence, allowing the advantageous roles to even out over time.

Early consensus termination. As an optimization, if a replica receives a VCBC proof for the same value for every participant, it knows in advance that the corresponding value is the only possible output of consensus and can return it immediately. However, it continues to run the consensus protocol in case other replicas do not receive the same view. This is particularly useful in distributed validators because replicas have a high chance of proposing the same value. This is because, for most validations, different inputs only occur when different replicas have a divergent view of the current state of the blockchain, which is rare. Note that this optimization differs from the first optimization described in Section 5, which refers to ABA instead of VCBC instances.

The current implementation consists of 5,000 lines of Go code, integrated as a subset of the large codebase of SSV.

Consensus layer for Filecoin subnets. We also implemented Alea-BFT as part of Mir/Trantor [45, 46], an experimental framework for distributed protocols, which is meant to become a new consensus layer for the subnets of Filecoin [18, 45]. This framework already supports the ISS-PBFT [52] protocol, and an upcoming implementation of a new protocol, apart from Alea-BFT. We implemented Alea-BFT in 4,000 lines of Go code and are currently merging it into the main repository [45]. It is nonetheless already freely available as open source [1]. The subprotocols used by Alea-BFT – ABA and VCBC – are implemented as independent modules, allowing their reuse by other protocols to be implemented within the same framework. In addition, we added support for threshold cryptography and BLS signatures within the framework, to be available to other protocols.

We improved the performance of this implementation through the parallel execution of agreement rounds. In Alea-BFT, the system can only order one batch of requests per agreement round, capping the system throughput to $BatchSize \times ABARate$. To overcome this limit, we allow multiple agreement rounds to make progress in parallel, but they buffer the delivery until it is guaranteed that the instances are delivered in order. However, we have to be careful so that this parallelization does not overload the network, while also remaining effective. To this end, we limit parallelization to the next N agreement rounds and restrict ABA execution as follows. Before all the preceding agreement rounds deliver, ABA instances are only allowed to make progress using the unanimity optimization and otherwise need to wait for their turn. Under this restriction, the eager execution of ABA instances only broadcasts up to two messages (INIT and FINISH), thus limiting its network impact.

9 Evaluation

We evaluated the three implementations of Alea-BFT under several scenarios. The following questions guide our evaluation. (1) How does the performance of Alea-BFT compare to other asynchronous BFT protocols across different configurations? (2) How robust is Alea-BFT to faults? (3) How do the real-world implementations of Alea-BFT compare to the previous protocols that they employed?

9.1 Experimental environment

Baselines. The research prototype of Alea-BFT was evaluated against two baselines. First, we compared it to our own implementation of HBBFT [40], the first protocol in the new generation of asynchronous BFT, and where, to obtain an apples-to-apples comparison, we started from the same codebase as Alea-BFT and tried to optimize the implementation of HBBFT to the fullest extent. Second, we compared to Dumbo-NG [24], which is the state-of-the-art asynchronous BFT protocol with outstanding performance (several-fold better than the direct competitors, according to their results [24]), and, in this case, we deploy their unmodified codebase. For the two real-world implementations, we used the protocols that those systems originally supported as baselines.

Setup. We deployed Alea-BFT in a cluster where the replica and client instances ran on machines equipped with AMD EPYC 7272 12-Core Processors. In addition, Docker was used to limit each replica's CPU usage to 4 cores and the Java VM was capped at 10GB of memory. These machines were connected to the same local network with 1Gb connections. To evaluate the effects of deploying Alea-BFT in a wide-area network, we emulate varying additional inter-replica latency using `netem`. Some deployment characteristics for the real-world prototypes differed whenever noted.

Clients submit requests in an open loop, and we vary the inter-request interval and the number of clients to increase the load. The payload size of the requests and responses is 256 bytes, aligned with the size of Bitcoin transactions (as noted and employed in previous work [40]). Each experiment runs for 2 minutes, and we repeat experiments 5 times and report the average. When measuring latency for Dumbo-NG, its implementation always loads the system, and therefore we would have to modify their codebase to measure the latency of an isolated request. As such, the latency measurements for Dumbo-NG represent the performance at an intermediate load generated by the implementation, where the system is not quiescent but also not as overloaded as during throughput measurements for the other protocols.

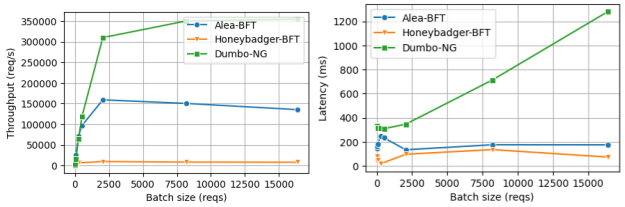
9.2 Performance under different parameters

We start by using our research prototype to measure the latency and throughput of Alea-BFT and the two baselines under different configurations and deployments, namely varying the batch size, the replication factor, and the inter-replica latency. When the sensitivity to one of these parameters is being evaluated, the remaining ones are fixed to a batch size of 1024, n =4 replicas (i.e., $f = 1$), and LAN (minimal inter-node latency), respectively.

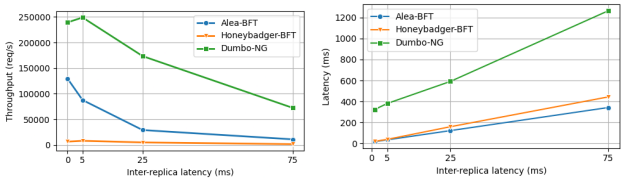
Figures 2a and 2b show the peak throughput and latency while varying the batch size. In this particular setting, the latency at the peak throughput is reported instead of the base latency (measuring an isolated request without any system load). This is because the latter would require issuing only one or a few requests, much less than the batch size. Thus, the measured latency would correspond to either slowly filling up the batch or triggering a batch timeout. The results show that Alea-BFT is competitive with the state of the art (Dumbo-NG) and that both are a significant improvement over their predecessor (HBBFT). While Alea-BFT cannot match the peak throughput of Dumbo-NG, they are both in the same order of magnitude (hundreds of thousands of txs/s), versus ≈15k txs/s for HBBFT. When considering latency, Alea-BFT outperforms Dumbo-NG across all tested batch sizes. We attribute these differences in throughput and latency to the choice of agreement primitive. In particular, Dumbo-NG uses MVBA, which allows for better throughput by accepting several batches at once, but the simpler ABA primitive of Alea-BFT enables a better latency under a comparable load. Note that the fact that Alea-BFT's throughput peaks earlier than Dumbo-NG's in this setting is mainly an implementation artifact – our codebase has an external open loop client that further saturates the network, unlike Dumbo-NG's.

Next, we use `netem` to evaluate the performance under different network conditions (LAN vs. WAN). Figures 2c and 2d show the peak throughput and latency when varying inter-replica latency. The results show that Alea-BFT has the lowest latency of all protocols while achieving a peak throughput in the tens of thousands of requests per second when the inter-node latency is under 25ms. HBBFT witnesses a similar degradation in latency to Alea-BFT because the critical path for a normal-case request execution is the same for both protocols, except for a single protocol step, which explains the slightly higher latency of HBBFT.

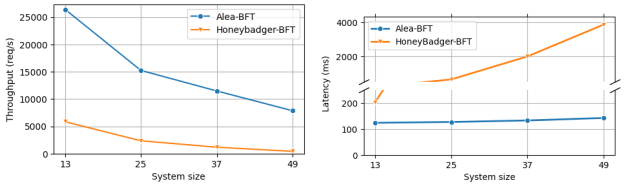
Finally, we scale out the experiments by increasing the number of replicas participating in the consensus. For these experiments, we use 13, 25, 37 and 49 replicas, and, in this case, we use `netem` to simulate a WAN environment, with a 75ms inter-replica latency, corresponding to an RTT of 150ms (approximating a cloud deployment). Furthermore, since the available setup forced some replicas to be co-located on the same machine, to ensure a realistic and uniform bandwidth availability, each instance's bandwidth was capped at 50Mb/s using a token bucket filter.



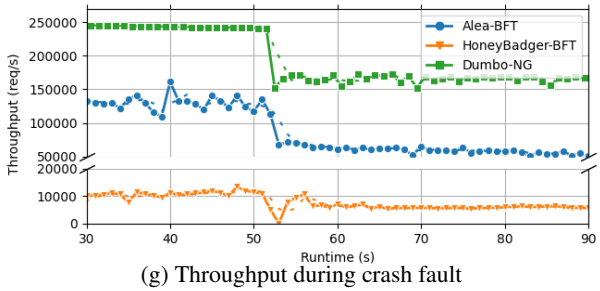
(a) Peak throughput vs batch size (b) Latency at peak throughput vs batch size



(c) Peak Throughput vs inter-replica latency (d) Base latency vs inter-replica latency



(e) Peak throughput vs system size (f) Base latency vs system size



(g) Throughput during crash fault

Figure 2: Prototype implementation evaluation

In this case, we were not able to configure the Dumbo-NG code to use the same replica group sizes as the ones we employed for the other two systems, which explains why there are only two curves. Our results show that Alea-BFT not only has superior throughput but also achieves very good latency in unloaded scenarios, due to the clients being able to predict the current leader and send requests to the replica that will drive the decision the fastest. On the other hand, in unloaded scenarios, HoneyBadger's clients need to contact $2f + 1$ replicas to ensure progress, meaning that, for a single request to go through, $2f + 1$ ABAs need to be executed.

9.3 Performance under faults

We compared the performance of Alea-BFT and the baselines in a scenario where one of the replicas crashes 50 seconds into the trace. We inject a crash fault instead of a protocol-specific Byzantine fault, for a direct comparison between protocols. The results in Figure 2g show that Alea-BFT and HBBFT suffer more with the crash of f replicas because they share the unanimity optimization described in

Section 5 (which cannot be used when a replica is unresponsive), but Dumbo-NG also takes a significant hit (around 30% of throughput) due to bandwidth wasted on the faulty replica.

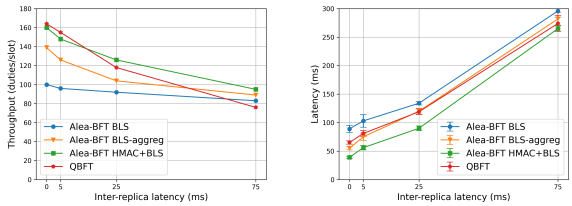
We also evaluate the performance under faults for the other implementations and present the results in the next section.

9.4 Real-world implementations

Ethereum distributed validator. We start this part of the evaluation with the implementation of Alea-BFT in the SSV distributed validator of Ethereum. In this case, the experimental methodology is constrained by the way that SSV operates, namely that the system progresses in a sequence of slots of fixed duration (12 seconds in Ethereum), and during each slot, each validator is assigned a set of duties (tasks such as block proposal and attestation). Thus, to measure the base latency, we set the number of duties per slot to 1 and measure the time to complete it, whereas throughput is measured by increasing the number of duties per slot until this metric peaks. Since the performance is not network-bound and the number of nodes is low, we did not use the bandwidth cap. We used a group of 4 replicas, with no added inter-replica latency, and batching is not applicable in this setting. We tested variants of Alea-BFT that use different message authentication methods and compared these to the existing QBFT-based codebase. In particular, we start with a direct comparison to QBFT (which, in the SSV codebase, uses BLS digital signatures without aggregation), and then add BLS aggregation to Alea-BFT. (This change could also be applied to QBFT, but this was avoided to keep the baseline as the existing codebase.) Then, we replace signatures with HMACs, which is possible in Alea-BFT but would not be directly applicable to QBFT, because of messages conveyed to all processes during round changes. In the case of HMACs, BLS is only used to verify the final VCBC signature and compute ABA's shared coin.

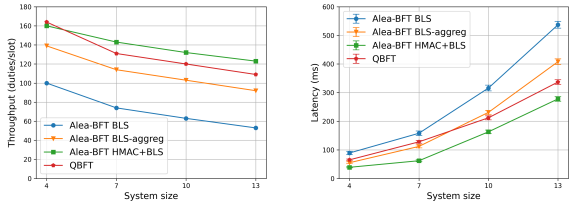
Figure 3 shows the performance of the SSV validator using different protocols. In these plots, the latency and throughput in the most basic setting can be determined by the leftmost point of Figures 3a and 3b. These results show that Alea-BFT with BLS aggregation and with HMACs has similar peak throughput and better latency than the previous codebase that uses QBFT. This highlights how designing Alea-BFT to have a small number of protocol steps, combined with the possibility of using HMACs that comes from not having the view change mechanism from partially synchronous protocols can lead to competitive performance in this setting. We attribute the slightly better throughput of QBFT to the leader-driven protocol allowing for exchanging a smaller number of messages. However, we see the overhead of Alea-BFT as a modest price to pay for not making partial synchrony assumptions.

The effects of varying the inter-replica latency is shown in the remainder of Figures 3a and 3b. The key takeaway is that, for all tested conditions, Alea-BFT closely follows QBFT in terms of base latency and peak throughput and, with the best choice of cryptographic primitives in place, Alea-BFT can



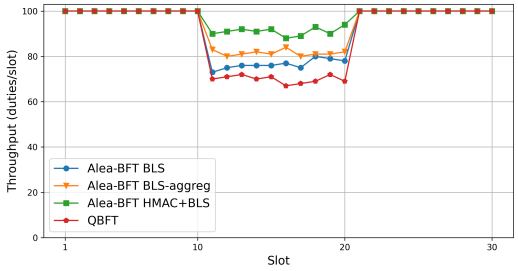
(a) Peak throughput vs inter-replica latency

(b) Base latency vs inter-replica latency



(c) Peak throughput vs system size

(d) Base latency vs system size



(e) Throughput during crash fault

Figure 3: Distributed validator deployment evaluation

even achieve lower latency values. Figure 3b also depicts the change of relative importance of cryptographic primitives as inter-replica latency varies – in a LAN environment, as the network delay is small relative to the cost of cryptography, the several variants have a very noticeable relative difference among them. However, as the inter-replica delay increases, this difference decreases in proportion.

Next, we measured performance as the group size increases (Figures 3c and 3d). Currently, in SSV, a validator can only employ 4, 7, 10, or 13 operators, as defined in its smart contract. In this experiment, as in the previous one, Alea-BFT's latency and throughput follow QBFT's, achieving lower latency and higher throughput values when using HMAC for point-to-point authentication and BLS digital signatures.

Finally, Figure 3e shows the results of an experiment where we crash one of the processes, chosen at random, at the beginning of the 11th slot in the run, then restart it in the 21st slot, and we plot the number of duties that are executed per slot throughout the trace. The results show that Alea-BFT is more resilient to this fault because of the principles behind its design: the fault will affect $1/4$ of the VCBC instances, but these rounds will quickly be skipped and replaced with productive work led by the other replicas. In contrast, QBFT waits for a timeout and a leader change protocol to complete, which slows down the entire system for that duration.

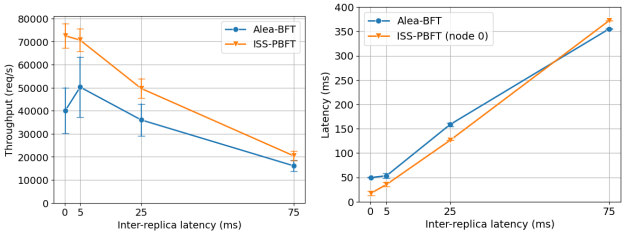
Consensus layer for Filecoin subnets. In the last part of this section, we evaluate Alea-BFT's integration into an experimental consensus layer [45, 46] for Filecoin subnets [18] against the existing implementation of ISS-PBFT [52] in the same codebase. ISS-PBFT uses the same parameters as in its original evaluation [52].

Base latency experiments use two co-located closed-loop clients per replica, ensuring incoming request queues are never empty. This was required because the implementation of ISS-PBFT stalls when there are no requests to propose. However, it has the downside of generating some load that may negatively affect latency. We only present results for replica 0 in the ISS-PBFT latency measurements due to an implementation issue that inflates latencies in other replicas, which would unfairly harm the baseline's performance. Peak throughput is measured in a configuration of $8 * B$ closed-loop clients co-located with each replica, where B is the batch size, which we empirically determined to maximize the throughput.

We begin by evaluating Alea-BFT's performance against ISS-PBFT when varying the inter-node latency. Figures 4a and 4b show that Alea-BFT closely follows the performance ISS-PBFT in wide-area settings, in terms of peak throughput and base latency. Furthermore, while Alea-BFT is initially limited to ≈40k requests/s and ≈50ms base latency, it becomes on par with ISS-PBFT when the limiting factor shifts from threshold cryptography to network latency.

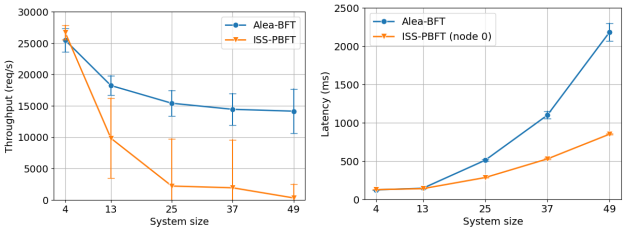
Additionally, we evaluated Alea-BFT's ability to scale against ISS-PBFT, which is relevant given that scalability is a key design goal in ISS. Figures 4c and 4d show peak throughput and base latency measurements for Alea-BFT and ISS-PBFT for a variety of system sizes. Regarding peak throughput (Figure 4c), Alea-BFT's throughput degrades gracefully as the system size increases, fully saturating the (bandwidth-capped) network. In contrast, ISS-PBFT degrades abruptly and stops processing requests altogether after a few seconds for $N = 49$. However, we believe this is an artifact of this implementation of ISS-PBFT, which reacts poorly under strained network conditions and is not intrinsic to the ISS-PBFT protocol. Regarding latency (Figure 4d), both protocols maintain near-constant base latency under system sizes up to $N = 22$, after which it begins to increase. In this case, ISS-PBFT has a lower latency than Alea-BFT because its multi-leader design allows requests to be processed as soon as they reach the PBFT primary replica, whereas in Alea-BFT we have to wait for the designated replica's turn to run its agreement round.

Finally, we studied the impact of crash faults on both Alea-BFT and ISS-PBFT. Figure 4e shows an execution trace of one Alea-BFT and one ISS-PBFT execution with the default settings, where a single replica crashes after 150s (and stays crashed). To aid evaluation, a dotted line was added to both curves, showing a moving average of the system's throughput across all repetitions. In this trace, we first observe a 15-second stall of ISS-PBFT after the crash, waiting for a timeout for the detection of the crashed replica, whereas Alea-BFT can continue uninterrupted (albeit at reduced throughput) thanks to its leaderless design. After this timeout expires, ISS excludes the crashed replica from the set of leaders and continues with a relatively small (≈20%) performance hit. However, Alea-BFT is penalized on two fronts – it both loses a replica proposing requests (like ISS) and the ABA unanimity optimization – leading to a reduction in throughput when compared to the system with all replicas functional.



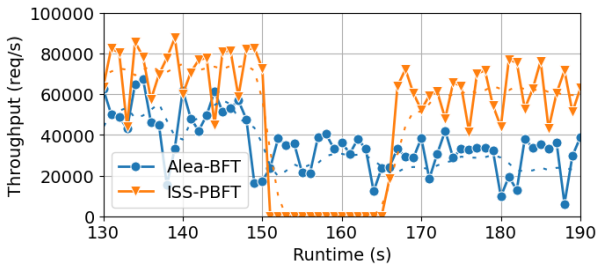
(a) Peak throughput vs inter-replica latency

(b) Base latency vs inter-replica latency



(c) Peak throughput vs system size

(d) Base latency vs system size



(e) Throughput during crash fault

Figure 4: Mir/Trantor deployment evaluation

10 Conclusion

In this paper, we presented Alea-BFT, a practical asynchronous BFT protocol with a design that combines simplicity with performance. Our experimental evaluation shows that Alea-BFT performs better than the top-performing Dumbo-NG in latency, offers comparable throughput, and is resilient to faults. Importantly, Alea-BFT is being adopted in the real world, namely by Ethereum distributed validators.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Zhaoguo Wang, for their helpful feedback. This work was supported by Fundação para a Ciência e a Tecnologia, projects UIDB/50021/2020 and PTDC/CCI-INF/6762/2020, and by the European Union's Horizon 2020 research and innovation programme, under grant agreement No 952226, project BIG.

References

[1] Mir codebase with Alea-BFT. https://github.com/abread/mir/tree/43a82f13b3f5353a80bdfe2fe2613daed0fbf710.

[2] Prototype implemenation of Alea-BFT. https://github.com/diogoantunes25/Alea-BFT.

[3] ssv codebase with Alea-BFT. https://github.com/MatheusFranco99/ssv.

[4] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[5] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, page 337–346, 2019.

[6] Enterprise Ethereum Alliance. EEA publishes QBFT blockchain consensus protocol. https://entethalliance.org/23-01-qbft-spec-version-1-released/.

[7] Ethereum Enterprise Alliance. QBFT blockchain consensus protocol specification v1. https://entethalliance.org/specs/qbft/v1/.

[8] Diogo S. Antunes, Afonso Oliveira, André Breda, Matheus Guilherme Franco, Henrique Moniz, and Rodrigo Rodrigues. Alea-bft: Practical asynchronous byzantine fault tolerance. arXiv:2202.02071 [cs.DC] https://arxiv.org/abs/2202.02071, 2022.

[9] James Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2):165–175, 2003.

[10] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous secure computations with optimal resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, PODC '94, pages 183–192, 1994.

[11] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, nov 1987.

[12] Vitalik Buterin. Post by @VitalikButerin on X. https://twitter.com/VitalikButerin/status/1588669782471368704.

[13] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2nd ed.)*. Springer, 2011.

[14] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology — CRYPTO 2001*, pages 524–541. Springer, 2001.

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

[16] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation, OSDI 1999*, pages 173–186, 1999.

[17] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009*, pages 153–168, 2009.

[18] Alfonso De la Rocha, Lefteris Kokoris-Kogias, Jorge M Soares, and Marko Vukolić. Hierarchical consensus: A horizontal scaling framework for blockchains. In *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 45–52, 2022.

[19] Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for byzantine agreement. *Journal of the ACM (JACM)*, 32(1):191–204, 1985.

[20] Sisi Duan, Michael K Reiter, and Haibin Zhang. Beat: Asynchronous bft made practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2028–2041, 2018.

[21] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, apr 1988.

[22] ethereum.org. Distributed validator technology. https://ethereum.org/en/staking/dvt/.

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

[24] Yingzi Gao, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo-NG: Fast asynchronous bft consensus with throughput-oblivious latency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1187–1201, 2022.

[25] Rati Gelashvili, Lefteris Kokoris-Kogias, Alberto Sonnino, Alexander Spiegelman, and Zhuolun Xiang. Jolteon and ditto: Network-adaptive efficient consensus with asynchronous fallback. In *Financial Cryptography and Data Security - 26th International Conference, FC 2022*, volume 13411 of *Lecture Notes in Computer Science*, pages 296–315. Springer, 2022.

[26] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous BFT closer to practice. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022*, 2022.

[27] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 803–818, 2020.

[28] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, 1994.

[29] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, PODC'21, page 165–175, 2021.

[30] K.P. Kihlstrom, L.E. Moser, and P.M. Melliar-Smith. The securering protocols for securing group communication. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume 3, pages 317–326 vol.3, 1998.

[31] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP 2007, pages 45–58, 2007.

[32] Oisín Kyne. The distributed validator protocol roadmap. Obol Network Blog. https://blog.obol.tech/roadmap-the-distributed-validator-protocol/, 2024.

[33] Leslie Lamport. How to write a 21st century proof. *Journal of Fixed Point Theory and Applications*, 11:43–63, 2012.

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

[35] Chao Liu, Sisi Duan, and Haibin Zhang. Epic: Efficient asynchronous bft with adaptive security. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2020)*, pages 437–451. IEEE, 2020.

[36] Yuan Lu, Zhenliang Lu, and Qiang Tang. Bolt-dumbo transformer: Asynchronous consensus as fast as the pipelined BFT. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022*, pages 2159–2173, 2022.

[37] Ethan MacBrough. Cobalt: Bft governance in open networks. *arXiv preprint arXiv:1802.07240*, 2018.

[38] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distrib. Comput.*, 11(4):203–213, oct 1998.

[39] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for wans. In *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, pages 369–384, 2008.

[40] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 31–42, 2016.

[41] Henrique Moniz. The istanbul bft consensus algorithm. arXiv preprint arXiv:2002.03613, 2020.

[42] Henrique Moniz, Nuno Ferreria Neves, Miguel Correia, and Paulo Verissimo. Ritas: Services for randomized intrusion tolerance. *IEEE transactions on dependable and secure computing*, 8(1):122–136, 2008.

[43] Achour Mostefaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with t< n/3 and o (n2) messages. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, PODC '14, pages 2–9, 2014.

[44] Ray Neiheiser, Miguel Matos, and Luís Rodrigues. Kauri: Scalable bft consensus with pipelined tree-based dissemination and aggregation. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 35–48, 2021.

[45] Matej Pavlovic. Mir – the distributed protocol implementation framework. https://github.com/consensus-shipyard/mir/blob/e100175138a4fd8947b6757452334698ee518967/README.md.

[46] Matej Pavlovic. Trantor: Modular state machine replication. https://github.com/consensus-shipyard/trantor-doc/blob/47dfc316a6d81604e1c567b823358f53fdfde4b4/main.pdf, 2023.

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

[48] Michael K Reiter. The rampart toolkit for building high-integrity services. In *Theory and practice in distributed systems*, pages 99–110. Springer, 1995.

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

[50] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 189–204, 2008.

[51] ssv.network. SSV into the future: Asyncrounous BFT protocols. https://ssv.network/blog/technology/ssv-into-the-future-asyncrounous-bft-protocols/.

[52] Chrysoula Stathakopoulou, Matej Pavlovic, and Marko Vukolić. State machine replication scalability made simple. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 17–33. ACM, 2022.

[53] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 135–144, 2009.

[54] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. DispersedLedger: High-throughput byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022*, pages 493–512, 2022.

[55] Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC'19, pages 347–356, 2019.