TÉCNICO LISBOA

Efficient Asynchronous Byzantine State Machine Replication

André Dias Duarte de Oliveira Breda

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Rodrigo Seromenho Miragaia Rodrigues
Eng. Henrique Lícias Senra Moniz

June 2024

Declaration

Acknowledgments

To my advisors Prof. Rodrigo Rodrigues and Henrique Moniz, thank you for the invaluable guidance over the course of this work, particularly regarding randomisation and scientific writing. To Diogo Antunes, Daniel Porto, Afonso Oliveira, Matheus Franco and my colleagues in INESC-ID, thank you for the fruitful discussions on optimisations, experimental setups and the various bugs found along the way. To Matej Pavlovic and the Protocol Labs team, thank you for your support in understanding Filecoin and Mir, and for the valuable insights on event-based systems. To Diogo Gaspar and Luís Fonseca, thank you for lending an extra pair of eyes to review my report. To the RNL administration, thank you for the quick help when the Cluster refused to collaborate. To my family and friends, thank you for (some of) the nagging, encouraging me during this time.

To each and every one of you – Thank you.

This work was supported by Fundação para a Ciência e a Tecnologia under the project PTDC/CCI-INF/6762/2020 and by the European Union's Horizon 2020 research and innovation programme, under grant agreement No 952226, project BIG. Part of this work was first published by the USENIX Association as "Alea-BFT: Practical Asynchronous Byzantine Fault Tolerance" [1].

Abstract

To address scalability difficulties, many blockchains resort to *sharding* in an attempt to distribute the load. In particular, the Filecoin network introduced a hierarchy of chains/networks [2], where *subnets* can abide by different rules as long as members agree. This approach allows employing traditional consensus algorithms, concretely BFT consensus [3], which can tolerate arbitrary faults. Traditionally, (BFT) consensus has been solved by resorting to network timing assumptions to circumvent the FLP result [4] (partially-synchronous consensus). However, networks are inherently asynchronous, thus these protocols sacrifice liveness, resulting in unavoidable performance faults. To address this issue, as an alternative to partially-synchronous consensus, researchers devised asynchronous consensus protocols, exploiting randomisation to solve consensus rather than timing assumptions. While early randomised protocols were impractical, recent proposals show promising performance. Even so, this class of algorithms has been relegated to theory and has seen little real-world use and evaluation. In this thesis, we implement Alea-BFT [1], a practical asynchronous BFT consensus algorithm, and integrate it in Filecoin, by building it on top of the Mir Framework [5]. This implementation bounds memory usage, which is not always considered when designing consensus protocols, and includes several optimisations, particularly for Alea-BFT's agreement stage, the most expensive component of the protocol. Furthermore, we comprehensively evaluate the protocol in real-world scenarios, comparing it with a state-of-the-art partially-synchronous BFT protocol – ISS-PBFT [6]. Our results show that Alea-BFT is competitive with ISS-PBFT in fault-free and crash-fault scenarios.

Keywords

Consensus; Blockchain; Asynchronous; Randomization; Byzantine Fault Tolerance

Resumo

Para colmatar os desafios de escala, muitos sistemas recorrem a *sharding* numa tentativa de distribuir a carga. Em particular, a rede Filecoin propôs organizar-se hierarquicamente, em várias subredes com regras independentes das demais acordadas pelos membros respetivos [2]. Crucialmente, esta abordagem permite aplicar algoritmos de consenso tradicionais, particularmente consenso BFT [3], que consegue tolerar faltas arbitrárias. Habitualmente, o problema de consenso (incluindo BFT), tem sido resolvido com assunções sobre o desempenho da rede (consenso parcialmente-síncrono). No entanto, redes são inerentemente assíncronas, e estas assunções, embora contornem o teorema FLP [4], sacrificam garantias de progresso levando a vulnerabilidades de desempenho inevitáveis. Além desta classe de algoritmos de consenso, a academia desenvolveu consenso assíncrono, que recorre a randomização em vez destas assunções. Ainda que propostas recentes de consenso assíncrono mostrem desempenho promissor, esta classe de algoritmos tem visto pouco uso e avaliação no mundo real. Assim, nesta tese implementamos o Alea-BFT [1], um algoritmo de consenso BFT assíncrono com bom desempenho, visando integrá-lo num sistema real, a rede Filecoin, construindo-o em cima do *framework* Mir [5]. Neste processo, tomamos cuidado de limitar o uso de memória, que nem sempre é considerado ao construir protocolos de consenso, e inclui várias otimizações, particularmente na componente mais cara do Alea-BFT (*agreement*). Por fim, comparamos esta implementação com um protocolo BFT parcialmente-síncrono de última geração – ISS-PBFT [6]. Os nossos resultados mostram que o Alea-BFT é competitivo com o ISS-PBFT em cenários sem faltas e com faltas *crash*.

Palavras Chave

Consenso; Blockchain; Assíncrono; Randomização; Falhas Bizantinas

Contents

List of Figures

List of Listings

Acronyms

ABBA	Asynchronous Byzantine Binary Agreement
ACS	Asynchronous Common Subset
API	Application Program Interface
BFT	Byzantine Fault Tolerance
BLS	Boneh-Lynn-Shacham
CFT	Crash Fault Tolerance
CPU	Central Processing Unit
DKG	Distributed Key Generation
DRBG	Deterministic Random Bit Generator
DSL	Domain-Specific Language
FHC	Filecoin Hierarchical Consensus
FIFO	First-In First-Out
FLP	Fischer-Lynch-Patterson
HBBFT	HoneyBadgerBFT
ISS	Insanely Scalable State Machine Replication
LAN	Local Area Nework
MVBA	Multi-Valued Byzantine Agreement
PBFT	Practical Byzantine Fault Tolerance
RPC	Remote Procedure Call
SB	Sequenced Broadcast
SMR	State Machine Replication
TCP	Transport Control Protocol
TOB	Total Order Broadcast

TSS	Threshold Signature Scheme
VCB	Verifiable Consistent Broadcast
WAN	Wide Area Nework

Glossary

Goroutine A task that is scheduled by the Go runtime concurrently with other tasks (goroutines). 29, 30, 62, 67

gRPC gRPC is a cross-platform RPC framework created by Google, which uses protobuf for message serialization and interface definitions. 42

libp2p libp2p is a modular and extensible networking stack for distributed (particularly peer-to-peer) applications created by Protocol Labs. 42

protobuf Protobuf, short for Protocol Buffers, is a message format and interface definition language developed by Google. It is designed to be language-neutral, platform-neutral, extensible, and space-efficient. 62, 101

Remote Procedure Call A communication abstraction for distributed systems where a process calls a procedure on another process transparently (as if the procedure was being executed locally). x, 28

1 Introduction

Contents

1.1 Motivation

A blockchain (or distributed ledger) is, at its core, an append-only log of unforgeable transactions. Users can broadcast their own transactions, which are then aggregated into blocks by validators (sometimes called miners), to be eventually included in the chain. The entire process of including a new transaction in the ledger, from the moment that it is created by a user to its inclusion in the chain, can be accomplished without relying on any central authority, as long as the validators are themselves independent. Due to this property, blockchain technology made decentralised currency and computation possible in projects like Bitcoin and Ethereum, but these systems suffer from a very low throughput.

To correctly implement the functionality of a distributed ledger, validators must agree on which block of transactions comes next in the chain, i.e., they must solve consensus. A particularly efficient way to do this is through the use of well-studied consensus (also known as agreement) protocols, which are a class of algorithms that fit the above specification. These have gained widespread adoption in the context of permissioned blockchains, where system membership is fixed, but are also being adopted in permissionless settings [9], leading to significant improvements in throughput [10].

At first glance, leveraging consensus can look simple, but our world is messy: the network is asynchronous (it may delay, reorder or lose packets) and nodes can fail. Furthermore, failures are not necessarily just crashes: the network may be controlled by an adversary [11], and participating nodes are not always trustworthy – in fact, this is an intrinsic part of the trust model underlying blockchains. Even if participants were to be trusted, nodes cannot: modern computers are the result of the integration of a myriad of systems, increasing the probability of faults from their complexity alone, and even in otherwise perfect software, environmental conditions alone can trigger hardware faults [12]. Therefore, for our system to be reliable, it is imperative to tolerate arbitrary (Byzantine) faults. In other words, we must have a solution for Byzantine consensus.

An intrinsic hurdle when devising a solution for consensus with faulty nodes is the need to work around the FLP impossibility [4], which states deterministic consensus is impossible in asynchronous systems under the presence of faults. To achieve this, many State Machine Replication (SMR) systems like PBFT [13], Tendermint [14], ISS [6] and Red Belly [15] assume a partially-synchronous network – one where the time to deliver a message is bounded, after some (unknown) point in the system execution. In this case, although the network is fundamentally asynchronous, it has enough periods of synchrony to allow this kind of algorithm to make progress and be performant during those periods.

While the partial synchrony assumption is enough to build consensus and SMR, systems built around it can be complex and hard to tune. These protocols usually rely on leaders [16], demanding view-change protocols to rotate them, which are not trivial. Additionally, leader-based protocols must deal with slow leaders, by rotating them often [17, 18] or running multiple parallel instances of consensus [6], increasing complexity. This combination results in a non-trivial trade-off when setting those timeouts between losing liveness for long periods when the leader is not responsive, versus risking a frequent and unnecessary leader rotation when nodes are working correctly but the network is slow.

Recent developments resulted in leaderless partially synchronous protocols like Red Belly [15], which ease the previous concerns. However, the non-trivial trade-off remains between keeping system latency as low as conditions allow versus tolerating momentarily slow nodes or network conditions.

Nevertheless, it is possible to solve consensus in an asynchronous settings if we stop restricting ourselves to deterministic algorithms. In particular, we can relax the termination property, by stating that the probability of the algorithm terminating is a value that can be as close as desired to one. This idea is not new: Fischer et al. [4] recognised this in the conclusion to their proof for the FLP impossibility, as well as the existence of multiple algorithms taking advantage of it. Unfortunately, these initial attempts came with a steep cost in communication and execution time. However, progress did not stop there, and several algorithms exist that offer an acceptable expected message complexity and number of rounds like HoneyBadgerBFT (HBBFT) [19] and Alea-BFT [1]. Alea-BFT is particularly interesting because it brought message complexity down to a value that is very close to the quadratic theoretical minimum, while also striving to be significantly simpler than its predecessors.

Overall, asynchronous consensus presents an opportunity to devise SMR protocols that can perform well with minimal to no tuning, even in varying network conditions, and in face of malicious participants.

1.2 Problem statement and contributions

Despite the recent advances in asynchronous consensus, this class of algorithms has seen little real-world usage and testing. To bring asynchronous consensus to the limelight, we believe more research is needed on the advantages and limitations of these systems, particularly regarding their competitiveness with existing partially-synchronous protocols. Therefore, a systematic study is required to evaluate the extent of these limitations and allow real-world applications to flourish.

In this thesis, we intend to address these shortcomings by building a production-ready implementation of Alea-BFT and integrating it into Filecoin under their hierarchical consensus initiative [2]. These objectives entail several modifications to the algorithm, adapting it not only to the physical constraints of the nodes that execute the protocol, whose memory and compute power are finite, but also to order Filecoin transactions in a subchain.

Furthermore, we evaluate Alea-BFT against a state-of-the-art partially-synchronous Byzantine Fault Tolerance (BFT) protocol – ISS-PBFT – in an apples-to-apples comparison to assess its competitiveness with traditional BFT protocols in various conditions.

Lastly, Alea-BFT is a recent algorithm that was never deployed in a real-world scenario, and we expect to uncover optimisation opportunities during our comprehensive evaluation in terms of performance, ease of use, and ease of implementation. Therefore, as a final contribution, we intend to build on this knowledge, improving the Alea-BFT algorithm to make it more robust, performant, and better aligned with the requirements and interfaces of real-world systems.

1.3 Document Outline

The remainder of this document is organised as follows. Chapter 2 presents related work on other consensus algorithms, Filecoin hierarchical consensus and the Mir framework. Chapter 3 describes the general architecture of our Alea-BFT implementation, including its integration into Filecoin. Chapter 4 goes over a set of optimisations that we implemented in Alea-BFT. Chapter 5 outlines our systematic evaluation methodology and presents our results. And finally, Chapter 6 summarises our findings and future work proposals.

2

Related Work

Contents

When building reliable systems, it is common to abandon the notion of a central server that responds to requests from clients, in favour of replicating it across multiple machines connected through a network [20]. This may be motivated by wanting to keep the system alive even if some of the machines fail, or even to ensure it remains correct in the presence of (possibly malicious) faults.

In this chapter we briefly explore the considerations involved in replication, focusing on SMR and its relationship with blockchains, threshold cryptography – a powerful primitive that simplifies protocols – and approaches for tolerating arbitrary (Byzantine) faults.

2.1 Replicated Systems

2.1.1 From SMR to Consensus

SMR [20] is a common method of achieving a replicated system. By modelling a system as a state machine, where changes to the system state are deterministically determined from the previous state and an operation, we can reduce the problem of replication to ordering the inputs to our state machine – the operations.

Each node starts in the same initial state, with the same state machine. Instead of receiving operations from clients directly, client requests first go through a Total Order Broadcast (TOB) component, which ensures that every node receives each request exactly once, in the same position in the total order.

The problem of achieving TOB, sometimes also called atomic broadcast, can then be solved by sequencing multiple instances of consensus – one of the most fundamental problems in distributed systems [3]. The consensus problem consists in having a set of nodes (sometimes referred to as processes) agree on a single value from their competing proposals.

There are a multitude of approaches to solving consensus. When considering a permissioned setting – where only authorised nodes can participate – solutions are often inspired by elections: replicas vote on which value they want to pick – sometimes in multiple rounds, to narrow down the set of candidate values – and eventually agree on a single value.

This work focuses on TOB protocols, leaving the details of the state machine to the application, but still allowing potential optimisations from tighter integration of the consensus component with the TOB protocol.

2.1.2 Byzantine Fault Tolerance

In order to tolerate faults, it is crucial to define what do we understand by correct and faulty nodes through a fault model.

One common fault model to consider is Crash Fault Tolerance (CFT), in which nodes are assumed to either be correct at all times, or correct until the moment they crash.

However, this model fails to capture hardware faults [12] or even malicious attackers that take control of nodes, which can cause nodes to deviate from protocol.

In the context of blockchains and cryptocurrencies, which are meant to establish trust by consensus among a set of non-mutually trusted parties, it is important not to assume anything about the behaviour of faulty nodes, or in other words, we must consider faults to be arbitrary. This is what in literature is known as BFT [3].

2.1.3 Relationship between SMR and Blockchains

SMR can support many applications, including distributed ledgers or, as they are commonly referred to in recent times, blockchains. However, classical SMR is not enough for many well-known blockchains such as Bitcoin and Ethereum as they are what is known as a **permissionless** (public) blockchain. In permissionless blockchains, where system membership can change at any time, quorum-based protocols break down, as any entity can create an unbounded number of identities and take control of the system, performing what is called a Sybil attack. Rather than relying on quorums, permissionless blockchains use agreement protocols that require nodes to spend finite resources, such as computing power or storage space, to participate in consensus.

In contrast, **permissioned** (private) blockchains can rely on classical SMR and quorum-based consensus protocols as system membership is tightly controlled: it only changes in explicit reconfiguration steps, and the organisation(s) controlling the blockchain can also restrict new configurations, making Sybil attacks a non-issue. Although this kind of blockchain is not fit for universal participation, it can achieve superior performance and is perfect for the centralised governance models typical of organisations and consortiums.

Despite Alea-BFT being a quorum-based TOB protocol, upon which is trivial to build an SMR system (see Section 2.1.1), this work allows its integration in a permissionless blockchain – Filecoin – using their hierarchical consensus framework, which are discussed in Section 2.5.

In the context of SMR, blockchain transactions are sometimes also referred to as operations, commands or client requests in the context of SMR, and transactions in the context of blockchains. From this point on, they are always referred to as transactions, as it is the nomenclature used in the Mir framework.

2.1.4 Solving Consensus under an Asynchronous Network

Aside from node faults, consensus algorithms must also deal with the inherent asynchronous nature of the network that connects nodes together.

Under an asynchronous network, each message can be delayed indefinitely, meaning they can be received in a different order from which they were sent or not received altogether.

As Fischer et al. showed in their famous FLP Impossibility result [4], it is impossible to devise a deterministic algorithm to solve consensus when the network is asynchronous.

However, researchers created multiple variants of the consensus problem that allow them to circumvent this result. We describe two of the most popular options: timing assumptions and randomisation.

2.1.4.A Partially-Synchronous Consensus

Partially-synchronous consensus algorithms introduce timing assumptions to exploit the fact that the network is mostly synchronous, apart from some unusual periods of asynchrony. In this specification of the consensus problem, protocols must remain correct (safe) in the presence of asynchrony, but only guarantee liveness during periods where network behaves well enough.

In formal terms, partial synchrony assumes that network delays are limited by some unknown bound, or that they are limited by a known bound after an unknown amount of time, known as the global stabilisation time.

Timing assumptions have been the most popular approach to tackle consensus, and are used in systems such as PBFT [13], ISS [6], Tendermint [14], and Red Belly [15], which we survey in Section 2.2.

This approach is common in permissioned blockchains such as Tendermint and Red Belly, which use the protocols with the same name, and even in some permissionless blockchains like Filecoin, which supports traditional consensus protocols like ISS under limited circumstances, which we discuss in Section 2.5.

2.1.4.B Randomised Consensus

Randomised consensus, sometimes called asynchronous consensus, relaxes the termination property of the classical consensus problem. Instead of demanding deterministic termination, it instead states that termination is a probabilistically certain event, i.e. the probability of terminating converges to one.

This approach was already known around the time the FLP impossibility was published, and several solutions were known and mentioned by the authors [21, 22]. Unfortunately these early solutions were inefficient, with several having an exponential expected number of steps when considering $F = \lfloor \frac{N-1}{3} \rfloor$ faulty nodes out of N total nodes. However, in the last decade, there was a surge in research on the topic of asynchronous BFT consensus, with several asymptotically efficient and practical solutions being proposed such as HBBFT [19], Dumbo [23] and variations [24, 25], and Alea-BFT [1], which we survey in Section 2.4.

2.2 Partially-Synchronous BFT

This section describes the ISS framework and Red Belly Blockchain, which are two partially-synchronous BFT protocols of interest to our work.

Traditionally, partially-synchronous BFT protocols rely on a leader to order transactions quicker, driving each consensus instance more efficiently by only allowing the leader to propose consensus values in each round (or view), but at the cost of performance when the leader is faulty. However, the protocols we survey in this section attempt to mitigate the cost of leader failure, either by using multiple leaders concurrently – in the case of ISS – or by not relying on a leader at all – in the case of Red Belly Blockchain.

By comparing Alea-BFT to these protocols, we are able to compare partially-synchronous vs asynchronous consensus without the confounding variable of leader reliance.

2.2.1 Insanely Scalable State Machine Replication

Insanely Scalable State Machine Replication (ISS) [6] is a modular SMR framework designed to help traditional leader-based TOB protocols scale, by allowing multiple leaders to concurrently order transactions, while avoiding redundant work ordering duplicate transactions.

The framework divides the transaction log into epochs, and each epoch into segments. Although epochs are strictly sequential, the transactions within each segment are ordered concurrently and combined deterministically (interleaved) to form the epoch. In order to avoid concurrent segments containing the same transaction, ISS partitions incoming transactions into buckets, and assigns each bucket to a single segment orderer.

Concurrent segment ordering allows ISS to scale, but without guaranteeing that the ordering protocol in each segment terminates epochs could never terminate, compromising the liveness property of consensus. To rectify this problem, the authors created a new primitive to order segments – Sequenced Broadcast (SB) – which wraps an existing TOB protocol and guarantees its termination. SB instances have a fixed leader, a fixed number of delivered transactions, and detect if the leader is quiet – a fault in a partially-synchronous setting – delivering a special empty value in this case.

By guaranteeing the termination of individual SB instances, the authors deal with potentially faulty leaders by rotating leaders/buckets across segments at the start of every epoch. With this construction, every bucket can be assigned to a segment with a correct leader infinitely often, and all its transactions are eventually ordered. For the rotation of leaders and buckets, the authors chose the policy of BFT-Mencius' [26], which excludes up to F nodes out of $3F+1$, minimizing the performance impact of faulty nodes deciding to slow down the SB instances they lead.

To evaluate ISS, the authors integrated various TOB protocols into it, and were able to achieve 37x, 56x and 55x improvements for PBFT, HotStuff and Raft, respectively, in fault-free executions on a 128 node deployment uniformly distributed across IBM's 16 data centres on Europe, America, Australia and Asia.

Finally, they investigated the performance of ISS-PBFT under the presence of crash-faults and Byzantine stragglers – Byzantine nodes that delay their proposals as much as possible without being detected as faulty. Regarding crash faults, ISS-PBFT improves upon its predecessor Mir-BFT [27][1] by keeping crashed nodes out of the leader set forever, and having a "more lightweight" crash fault recovery. Regarding Byzantine stragglers, the throughput of ISS-PBFT drops by 85% with one straggler, and by 10% with ten stragglers. The authors note that straggler-associated slowdown is inevitable to any SMR system until they are removed from the leader set, and that a more sophisticated leader rotation policy that dynamically detects and excludes stragglers could mitigate this issue.

Recently, the ISS implementation has undergone a refactoring effort to improve the Mir framework as a prototyping tool for new consensus protocols and facilitate its usage as a consensus layer for real-world systems like the Filecoin network, which we go over in detail in Section 2.5. Unfortunately, this means some parts of ISS have not been fully re-implemented yet. Namely, only PBFT ordering is implemented (but neither HotStuff nor Raft are), and the partitioning of transactions into buckets is not yet implemented, with all transactions being assigned to a single common bucket in each node.

2.2.2 Red Belly

The Red Belly Blockchain [15] aims to be a blockchain that scales to hundreds of participants across the world, open to transactions from untrusted nodes, while remaining secure, guaranteeing that no double-spending occurs. The authors were particularly concerned with avoiding strong synchrony assumptions (and related attacks), the performance penalty of faulty leaders, and the cost of transaction verification.

Rather than agreeing on individual sets of transactions (blocks), proposed by individual nodes, Red Belly nodes agree on a *superset* (*superblock*), containing proposed transactions from least $N-F$ nodes (out of $N \geq 3F+1$ total nodes, where up to F nodes can fail arbitrarily). By rotating the proposer whose set of transactions is added to the *superset* first, Red Belly avoids transaction censorship.

Red Belly's *supersets* are decided using a leaderless TOB protocol, which is uncommon in the partially-synchronous world, and bears some similarities to HBBFT (which we discuss in Section 2.4.1). We focus our presentation on the TOB protocol as it is the most relevant part of Red Belly to our work.

The Red Belly blockchain progresses in rounds, and decides the contents of a new *superset*. At the start of each round, nodes propose a set of transactions using a verified reliable broadcast primitive. This broadcast also includes Red Belly's sharded transaction verification protocol, which partitions nodes into $(F+1)$ primary and $(N-F-1)$ secondary verifiers. Primary verifiers verify transactions immediately,

[1] Mir-BFT should not to be confused with the Mir framework, which was the result of modularising the Mir-BFT code base for implementing ISS.

while secondary verifiers optimistically wait for the primary verifiers to broadcast their result, but resort to verifying transactions themselves after a timeout.

After broadcasting transactions, nodes must agree on which sets were successfully broadcast and are to be delivered. This is accomplished with one binary consensus instance per proposer node/set of transactions.

As broadcasts complete, nodes vote to deliver them in the corresponding binary consensus instance. After receiving broadcasts from $N-F$ nodes, and a timeout expiring, Red Belly nodes assume they have received all broadcasts for the round, and vote against delivering transactions from the remaining nodes. By always waiting for a timeout, Red Belly likely avoids wasting sets of transactions that take longer to receive than the first $N-F$ sets, but still arrive within the timeout, which increases with the age of the oldest unordered transaction.

After all binary consensus instances decide, the transactions from each accepted broadcast are combined into the *superset*, by concatenating all decided sets of transactions, and removing conflicting transactions. This concludes the round and allows nodes to start a new one.

The authors evaluated Red Belly extensively, spanning various configurations of hundreds nodes on Amazon EC2, both in single data centre and multi-region deployments, and both in low-end and high-end machines. On high-end machines, Red Belly reached 660k tx/s with 260 nodes.

To assess the impact of signature verifications in blockchain performance, the authors turned to low-end machine configurations, where the node's Central Processing Unit (CPU)s becomes the bottleneck. Red Belly's sharded verification allowed it to not only sustain throughput as the number of nodes increase, but also surpass the rate of transactions that could be verified locally. The other systems under study, HBBFT and a blockchain based on PBFT, were unable to sustain a throughput over 4k tx/s, and HBBFT reached latencies of over 60s when scaling past 100 nodes. The authors attributed the poor scalability of HBBFT under these conditions to the minimum quadratic bound on verifications, related to its use of erasure coding.

Finally, Red Belly was evaluated under the presence of Byzantine faults, particularly Byzantine proposers that fail to disseminate their transactions across all nodes. It maintained comparable latencies to the fault-free case (920ms vs 2300ms), but suffered a sharp increase in message complexity (538MB vs 2622MB). In any case, the authors note that the message complexity is still lower than HBBFT's (3600MB), which is not sensitive to this fault by design, due to its use of erasure coding.

Unfortunately, Red Belly is not open-source, making experimental comparisons to our work challenging.

2.3 Threshold Cryptography

Before moving on to the asynchronous BFT protocols, we need to provide background on threshold cryptography – a cryptographic primitive that all asynchronous BFT protocols we present employ. Computer systems leverage cryptography to ensure data confidentiality, authentication or integrity. (t, N)-Threshold cryptography generalises these constructs to a group setting by splitting the secret key and all computation that involves it among all group members. Performing operations that involve the secret key requires the cooperation of at least t out of N group members, and the cryptosystem remains secure even if $t-1$ group members are compromised. [28]

In the context of Byzantine fault tolerance, threshold cryptography is interesting because it can enforce the participation of a quorum of nodes and tolerate faults by design when performing sensitive operations. Concretely, it can be used to construct common random sources [29], prevent targeted censorship of consensus proposals, and alleviate the complexity of consensus algorithms (see Section 2.4).

This section presents two applications of threshold cryptography – threshold decryption and threshold signatures – and the challenges associated with their usage, namely their setup (key generation and distribution). We only consider non-interactive threshold schemes, that is, threshold schemes that do not require non-trivial network communication. This restriction excludes the popular ECDSA and EdDSA signature schemes, for which we could not find any non-interactive threshold (signature) adaptations. Furthermore, we focus on threshold signatures as opposed to threshold decryption, as Alea-BFT – the protocol under study – requires threshold signatures.

2.3.1 Threshold Decryption

A (t, N)-threshold decryption protocol requires at least t parties to collaborate to decrypt ciphertext on behalf of a group of N entities [28]. By leveraging threshold decryption, information can remain confidential (as ciphertext) until the threshold of decrypting parties is reached and only then be revealed to the group. In this scenario, group members produce a decryption share from the ciphertext using their private key share, representing a partial computation of the decryption process. Afterwards, t decryption shares can be combined to produce the original plaintext.

Additionally, some threshold decryption schemes have a robustness property, which guarantees that any ciphertext, even if adversary-controlled, has at most one valid corresponding plaintext.

2.3.2 Threshold Signatures

A (t, N)-Threshold Signature Scheme (TSS) is a (t, N)-threshold variation of traditional cryptographic signature schemes, which facilitate the creation and verification of cryptographic non-repudiable proofs of data authenticity – digital signatures [28]. This construction must guarantee two basic security properties:

- **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.

Traditionally, asymmetric TSSs expose four different operations (excluding the setup, which we discuss in Section 2.3.3), which we refer to by the following names throughout the thesis:

- `SignShare(PrivateKeyShare, Data) -> SigShare`, which constructs a signature share from the local nodes's private key share for the given data.
- `Recover(PublicKey, Data, Set[SigShare]) -> Sig ∪ {Error}`, which reconstructs the full signature from signature shares of at least t nodes for the given data and group public key. If all t signature shares are valid, the resulting full signature is guaranteed to be valid.
- `VerifyShare(PublicKey, Data, SigShare) -> {True, False}`, which verifies if a signature share is valid for the given data and group public key.
- `VerifyFull(PublicKey, Data, Sig) -> {True, False}`, which verifies if a (full) signature is valid for the given data and group public key.

There are several threshold signature algorithms, of which we highlight RSA and BLS. Threshold-RSA is a non-interactive TSS based on RSA, but, like RSA, it suffers from relatively large signature size requirements. Although ECDSA is a popular alternative to RSA thanks to its small signature size with equivalent security guarantees, we could not find any non-interactive threshold ECDSA algorithm [30], so we do not consider it in this thesis. Finally, BLS also provides short signatures and a simple non-interactive threshold BLS scheme exists, but signature verification is slower than ECDSA and RSA.

2.3.3 Threshold Key Generation and Distribution

There are multiple ways of setting up a threshold cryptosystem. A simple method is to trust a third-party dealer, which is responsible for generating and distributing keys to each group member. However, practical applications of threshold cryptography call for Distributed Key Generation (DKG), where group members can agree on group keys without a trusted dealer and without any member being able to deduce more than its share of the secret key from the interaction.

DKG is not trivial and is equivalent to consensus when considering non-synchronous communication among nodes. That said, recent proposals include asynchronous and partially-synchronous DKG capable of tolerating Byzantine faults with reasonable asymptotic efficiency [31–33]. For instance, the latest proposal we reviewed for asynchronous DKG achieves expected near-cubic communication complexity and expected cubic computational complexity per node [31]. Due to the effort involved in implementing DKG protocols, this component is left as future work.

2.4 Asynchronous BFT

In this section, we present three asynchronous BFT protocols. We begin by summarising HoneyBadgerBFT, the first practical asynchronous BFT protocol, followed by two of its successors, Dumbo-NG and Alea-BFT (the protocol under study).

2.4.1 HoneyBadgerBFT

HBBFT [19] addresses the demand for BFT protocols that can operate at the large scale required by cryptocurrencies. The authors argue that such applications, where participants are distributed across the globe and often mutually distrustful, cannot critically rely on network timing assumptions for liveness, and instead should turn to asynchronous protocols. Previously, this class of protocols was considered impractical due to their high expected communication complexity per transaction, but HBBFT lowered this expected complexity to the asymptotically optimal $O(N)$ bits per transaction, making it the first practical asynchronous BFT protocol.

This protocol builds upon the insight of previous work that the problem of Total Order Broadcast can be reduced to Asynchronous Common Subset (ACS), which is the problem of agreeing on a set of inputs to output, instead of a single input like in consensus. For this reason, ACS is sometimes called *vector consensus*. The resulting set is guaranteed to contain the inputs of at least $N-F$ proposer nodes.

To construct TOB from ACS, we can repeat instances of ACS in rounds. In each round, every node inputs a message they want to deliver into ACS, which decides a set of ($N-F$ or more) messages (transactions), and then delivering the resulting set of messages. For efficiency, instead of having each node propose individual messages (transactions), they propose batches containing multiple messages, and can order all the messages in a batch in one round.

The two key contributions of HBBFT are the elimination of redundant work among nodes without compromising fairness (avoiding censorship), and efficiently implementing ACS, without resorting to an Multi-Valued Byzantine Agreement (MVBA) protocol.

Firstly, to prevent transaction censorship, HBBFT encrypts transactions using a threshold decryption primitive [2] before inputting them to ACS, and ensures they are only decrypted after ACS terminates – threshold decryption protocols require a threshold of nodes to cooperate to decrypt the transactions, but

[2] HBBFT authors often refer to this primitive as threshold encryption, but it is more accurately described as threshold decryption, as the threshold is required for decrypting, not encrypting.

not encrypt (see Section 2.3.1). This way, an adversary cannot know the contents of a transaction before it is delivered by the protocol and therefore has no way to choose which transactions to try to censor.

Secondly, instead of using MVBA to realise ACS like its predecessor, HBBFT uses a combination of reliable broadcast and binary agreement primitives. In particular, all nodes disseminate their proposals using a reliable broadcast primitive, and, after receiving $N - F$ broadcasts, vote on the delivery of all N proposals in N instances of a binary agreement protocol – 1 if it was received, 0 otherwise. Afterwards, all the proposals for which the corresponding binary agreement instance delivers 1 are delivered as the result of the ACS protocol.

Despite this simpler approach to ACS being known for longer than MVBA, the reliable broadcast primitive was too costly for it to be practical ($O(N^2 * |v|)$ bits sent per reliable broadcast, where $|v|$ is the payload size). By leveraging a recent solution to reliable broadcast based on erasure codes, the construction used in HoneyBadgerBFT becomes practical as it is asymptotically equivalent to a one-shot broadcast in communication complexity ($O(N * |v|)$) for large enough payloads.

The authors evaluated HBBFT in a multi-region deployment on Amazon EC2, and measured a throughput of over 20k tx/s on a 40 node deployment, and over 1.5k tx/s on a 104 node deployment. When compared to PBFT, HBBFT was not only better able to sustain throughput, but was also CPU-bound, whereas PBFT was network-bound. They note the loss in HBBFT throughput is due to the large number of signature verifications ($O(N^2)$), and the single-threaded implementation.

Additionally, they observed that HBBFT was able to operate in an extremely challenging environment – the Tor network – where network latency is not only high, but also extremely variable. In an 8-node deployment, with an average network latency of 12 seconds and variance over 2000 seconds, HBBFT was able to sustain a throughput of over 800 tx/s without any additional tuning.

2.4.2 Dumbo-NG

Dumbo-NG is the latest work of a long line of research that stemmed from the original Dumbo protocol [23], which was itself an improvement on HBBFT. It is a state-of-the-art asynchronous TOB protocol that aims to attain high throughput without significant latency sacrifices and be resilient to censorship.

The authors noted that the ACS framework split the bandwidth intensive proposal dissemination from the latency-sensitive agreement, and that previous work focused on improving the efficiency of these stages individually. Unfortunately, these earlier solutions sacrificed latency for throughput by relying on large batch sizes, are prone to censoring transactions from the F slowest nodes, demand more (potentially limitless) computational resources, or asymptotically more rounds of communication to avoid censorship. To address these issues, Dumbo-NG uses a design where multiple batches from the same proposer can be decided in the same agreement round, and where any proposal from an honest node is eventually delivered. The resulting protocol is similar to Alea-BFT (which we present in the next section), in that it also features concurrent proposal dissemination with agreement.

In this protocol, all nodes constantly disseminate proposals with a fixed batch size. Batches are identified by the proposer node's identifier and a sequence number – *slot* – assigned by the proposer. To disseminate each batch, the protocol uses a form of Verifiable Consistent Broadcast (VCB). This primitive relies on threshold signatures to ensure that two honest nodes never see different batches for the same proposer-*slot* pair and to produce a *proof of retrievability*, that allows any node to verify that a batch was disseminated to enough nodes to always be retrievable. As an additional optimisation, the *proof of retrievability* is sent in the first message of the next broadcast.

Simultaneously, the always-running agreement component waits for new batches from $N-F$ different nodes before starting a new round. In each round, nodes use an MVBA primitive to decide what batches are to be delivered. The input to this primitive is the last received slot from each node, accompanied by the corresponding *proofs of retrievability*. The global predicate of the MVBA was fine-tuned to enforce safety and liveness of the protocol: the output's *proofs of retrievability* must be valid and the output must contain new batches to deliver from at least $N-F$ proposers. Crucially, this MVBA primitive has guaranteed *quality*: the probability of the decided value having been input by an honest node is $1/2$, which ensures that all correctly disseminated proposals are eventually delivered, since all correct nodes eventually propose them to the MVBA.

However, it is possible for the agreement component to decide to deliver transactions that are not yet available on all nodes, either due to bad network conditions, or to Byzantine proposers not fully disseminating their batches. To solve this problem, the authors introduced an additional sub-protocol that is used to request batches from other nodes. Nodes in need of help broadcast a `CALLHELP` message, and receive in response `HELP` messages containing the required information to reconstruct the missing transactions. It uses erasure coding and Merkle trees to keep communication complexity under control, bringing each `HELP` message to $O(1/N)$ of the batch's size.

The authors evaluated a single-threaded Python implementation of Dumbo-NG in an AWS deployment that spanned 16 regions. In this Wide Area Nework (WAN) setting, with 16 nodes, Dumbo-NG achieved over 100k tx/s of throughput with a relatively small batch size (1k tx/batch), while maintaining latencies of under 2 seconds. This stable latency as throughput increases was identified as one of the key advantages of Dumbo-NG.

As batch sizes increased to 5k tx/batch, throughput peaked at over 150k tx/s, while latency increased by less than 0.5 seconds. Previous work in the Dumbo family of protocols, which are also based on the work of HBBFT, required much larger batch sizes to reach their peak throughput (over 10k txs per batch), while suffering from much higher latencies (over 5 seconds in peak throughput), and never reaching the throughput of Dumbo-NG.

The authors also investigated the performance of Dumbo-NG's sub-protocols. They validated that their MVBA primitive's performance (latency) did not depend on network bandwidth capacity, and that their broadcast component could nearly saturate the network with small batch sizes.

Finally, they refer some important aspects that must be taken into account to make their prototype implementation production-grade, and argue that strong validity is a crucial property for censorship resilience.

At a more practical level, we noticed in our experimental evaluation that this protocol requires a small modification to make progress under low or uneven loads: honest nodes must occasionally broadcast empty batches when they have no transactions to deliver, to ensure that other honest nodes that do have undelivered batches are not perpetually stuck waiting for a new MVBA to start (which requires $N-F$ nodes to have undelivered batches). Similarly, the broadcast of a batch only completes when the broadcast of the next batch starts, due to their broadcast protocol piggybacking the batch's signature in the first message of the **next** broadcast, which also presents a liveness problem under low or uneven loads. Occasionally broadcasting an empty batch also solves the liveness issue with Dumbo-NG's broadcast component.

2.4.3 Alea-BFT

Alea-BFT [1] is an asynchronous protocol, also based on the work of HBBFT, which aims to reap the benefits of concentrating work on specific nodes while avoiding the performance penalty of over-reliance on a leader. To this end, nodes locally order their proposed transactions in a queue, replicate it to other nodes, and continuously agree on which queue's head – if any – is to be delivered (globally ordered) next. By leveraging this architecture, it is able to perform very close to the optimal asymptotic bounds regarding time, message and communication (expected) complexity. Concretely, the protocol incurs an expected time complexity of $O(\sigma)$, expected message complexity of $O(N^2 * \sigma)$, and expected communication complexity of $O(N^2 * (|m| + \sigma\lambda))$, where λ is the size of a threshold signature, $|m|$ the size of a transaction, and σ the number of agreement rounds required for delivering a broadcast transaction, which the authors argue is a very small constant, and close to 1 in practice.

The protocol is divided into two components: broadcast and agreement. The broadcast component is responsible for appending new (batches of) transactions to the node's queue, and managing local and remote queue replication. Crucially, the broadcast component ensures that no two honest nodes see different transactions in the same queue slot. The agreement component is responsible for deciding when to deliver the head of each queue, using a consensus primitive.

We begin by presenting the two primitives used by Alea-BFT – VCB and ABBA [3] – followed by the choice of common coin and, finally, how the broadcast and agreement components of the protocol are

[3] In the original description of Alea-BFT, Verifiable Consistent Broadcast (VCB) is abbreviated as VCBC and Asynchronous Byzantine Binary Agreement (ABBA) as ABA. We use the new abbreviations throughout this document for consistency with the terms used in the implementation.

combined in Alea-BFT. This description of the protocol does not include the transaction deduplication mechanism which is impractical to implement as described, since it relies on storing the full set of delivered transactions indefinitely.

2.4.3.A Verifiable Consistent Broadcast (VCB)

VCB is a protocol for one node – the broadcast leader – to send a message to all nodes. Despite not guaranteeing that all correct nodes deliver the message when the leader is Byzantine, it does ensure that all nodes that deliver a message deliver the same one. Additionally, correct nodes that deliver the message can produce a succinct proof that allows other nodes to do the same safely and immediately.

Formally, a VCB protocol satisfies the following properties [34]:

- **Validity** If a correct sender broadcasts m, all correct nodes eventually deliver m.
- **Integrity** Correct nodes deliver at most once.
- **Origin Integrity** If a correct node delivers m, it was sent by the sender.
- **Consistency** If two correct nodes deliver m and m', then $m = m'$.
- **Verifiability** If a correct node delivers m, it can produce a new message $M = (m, \sigma)$ that it may send to other nodes, such that any correct node that receives M and hasn't delivered anything can safely deliver m.
- **Proof Succinctness** The size of the proof σ carried by M is independent of the size of the message m.

We will now describe the VCB protocol used by Alea-BFT. This description differs from the ones presented by Antunes et al. [1] and Cachin et al. [34] in that it includes an optimisation for the `ECHO` and `FINAL` messages. In the original description, every protocol message includes the broadcast value, which is redundant as messages must be tagged with their unique VCB instance identifier. We instead only send the broadcast value in the `SEND` message and assume all further communication refers to the broadcast value contained in this message. With this optimisation, assuming the leader is honest, the broadcast value is only sent once to each node. Note that VCB remains safe under the presence of Byzantine nodes: the broadcast value included in the `SEND` message is the only acceptable value to use in any further messages, and any messages that could have a different broadcast value in the original protocol would already be rejected. The existing validation methods for the full threshold signature (broadcast by the leader) and signature shares (sent by followers) are sufficient to detect and exclude nodes attempting to sign a different broadcast value.

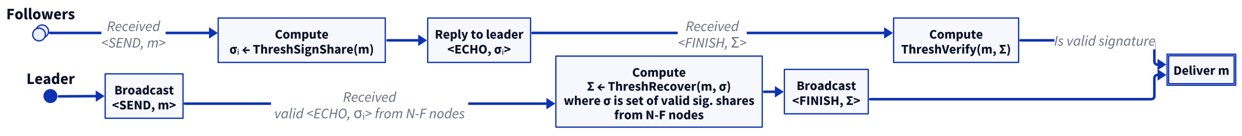

Followers
Received <SEND, m>
Compute σᵢ ← ThreshSignShare(m)
Reply to leader <ECHO, σᵢ>
Received <FINISH, Σ>
Compute ThreshVerify(m, Σ)
Is valid signature
Deliver m
Leader
Broadcast <SEND, m>
Received valid <ECHO, σᵢ> from N-F nodes
Compute Σ ← ThreshRecover(m, σ) where σ is set of valid sig. shares from N-F nodes
Broadcast <FINISH, Σ>

Figure 2.1: Activity Diagram of the VCB Protocol used in Alea-BFT

This protocol relies on threshold signatures for directly enforcing several of the previously mentioned properties, namely Consistency, Verifiability and Proof Succinctness. Note that, for simplicity, the broadcast leader also acts as a follower, and broadcasting a message to all nodes includes sending it to itself. We present an activity diagram of this protocol in Figure 2.1.

The protocol begins with the leader sending `<SEND, m>` to all nodes (followers) containing m – the message to be broadcast. Upon receiving the `SEND` message, each follower node i computes its threshold signature share σ_i for m and sends it to the leader in an `<ECHO,` σ_i`>` message. Importantly, honest nodes ignore any future `SEND` messages, ensuring that they never sign more than one message for the same VCB instance.

Upon receiving at least $N-F$ `ECHO` messages from different nodes, the leader reconstructs the threshold signature Σ from the set of collected shares σ, and sends it to all nodes in the form of a `<FINAL,` Σ`>` message.

Finally, nodes eventually receive the `FINAL` message, verify that Σ is a valid signature for m, and deliver m. The leader can bypass signature verification since it can trust itself to produce a valid one, delivering m immediately after broadcasting `FINAL`.

By relying on threshold signatures for consistency, the protocol achieves remarkable efficiency in communication: it has linear message ($O(N)$) and communication ($O(N|m|)$) complexity, and it terminates in just three rounds of communication.

2.4.3.B Asynchronous Byzantine Binary Agreement (ABBA)

Binary consensus is a variant of consensus where the value to decide is a single bit – zero or one. We specifically require an ABBA protocol: binary consensus that works in an asynchronous setting and tolerates Byzantine faults.

Formally, binary consensus protocols guarantee the following properties [35]:

- **Agreement** If two correct nodes decide b and b', then $b = b'$.

- **Validity** If all correct nodes input b, then any correct node that decides, decides b.

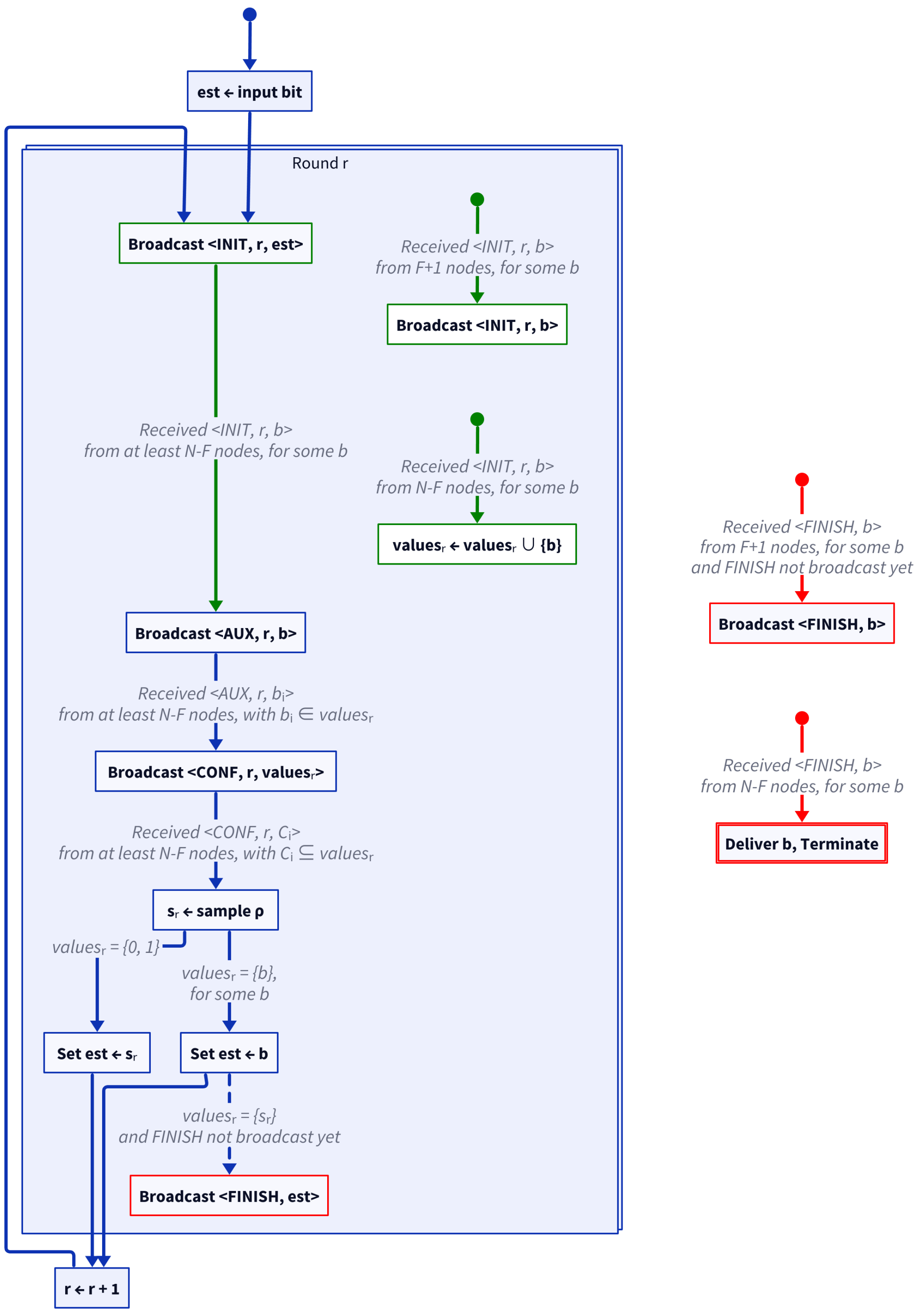
The previous list lacks a **Termination** property, which is crucial to any algorithm, particularly consensus algorithms. If we were operating under partial synchrony assumptions, termination would mean that every correct node eventually decides, but Fischer et al. have proved this combination of requirements impossible to satisfy. To circumvent the FLP impossibility [4] result with randomisation, termination is reframed to be probabilistic [35]:

- **P-Termination** The probability that a correct node decides after r rounds approaches one as r approaches infinity.

We will now briefly describe the ABBA protocol used in Alea-BFT, which is the one from Mostéfaoui et al. [35] with a fix for a liveness issue [36] also present on Cobalt's binary agreement [37]. It provides optimal fault resilience, and optimal expected asymptotic complexity in time ($O(1)$), messages ($O(N^2)$) and communication ($O(N^2\lambda)$, where λ is the overhead associated with the common coin). This protocol relies on an $F+1$-strong common coin ρ, meaning its value can only be revealed after $F+1$ nodes (i.e., at least one honest node) attempt to sample it. Since we are interested in implementing this protocol, we describe it from the point of view of a node and not the overall system (e.g., we refer to $values_r$ and not $values_r^i$, removing the reference to a specific node i). The protocol is represented as an activity diagram in Figure 2.2. In this diagram color is used to group related functionality together. Additionally, there exist multiple entry points (solid color circles) and a single dashed connection, which should be interpreted as concurrently executing.

The Cobalt ABBA protocol progresses in rounds and can deliver before a value is input by a local node. Delivery and termination happen after receiving a Byzantine quorum (at least $2F+1$ nodes) of `FINISH` messages. Additionally, nodes help the protocol converge faster by helping to disseminate `FINISH` messages from honest nodes by broadcasting `<FINISH, b>` after receiving it from $F+1$ nodes if the node hasn't sent any `FINISH` message already (refer to red sections in Figure 2.2).

Each ABBA round r begins by broadcasting the current proposal est in an `INIT` message, which can be different from its original proposal after a round ends in a tie, to ensure the protocol eventually decides. The set $values_r$ tracks which bits reach a Byzantine quorum of `INIT` messages for round r and is continuously updated during the entire round (refer to the green sections in Figure 2.2). Afterwards, the protocol uses the `AUX` and `CONF` messages to confirm strong support for any single bit. If there is strong support for bit b, est is set to b. Furthermore, if this bit matches the result of sampling the common coin for this round ($s_r = b$), the node broadcasts `FINISH`, if it hasn't done so already, with bit b. Otherwise, if we are tied between one and zero (i.e. $values_r$ contains both), we set est to s_r – the random value we sampled earlier – which helps correct nodes converge to the same proposal. Lastly, a new round begins, and all the state relating to the previous round may be discarded. Eventually, correct nodes converge to the same proposal, and Byzantine nodes become unable to influence the execution of the protocol.



est ← input bit
Round r
Broadcast <INIT, r, est>
Received <INIT, r, b> from F+1 nodes, for some b
Broadcast <INIT, r, b>
Received <INIT, r, b> from at least N-F nodes, for some b
Received <INIT, r, b> from N-F nodes, for some b
values_r ← values_r ∪ {b}
Received <FINISH, b> from F+1 nodes, for some b and FINISH not broadcast yet
Broadcast <FINISH, b>
Broadcast <AUX, r, b>
Received <AUX, r, b_i> from at least N-F nodes, with b_i ∈ values_r
Broadcast <CONF, r, values_r>
Received <FINISH, b> from N-F nodes, for some b
Deliver b, Terminate
Received <CONF, r, C_i> from at least N-F nodes, with C_i ⊆ values_r
s_r ← sample ρ
values_r = {0, 1}
values_r = {b}, for some b
Set est ← s_r
Set est ← b
values_r = {s_r} and FINISH not broadcast yet
Broadcast <FINISH, est>
r ← r + 1

Figure 2.2: Activity Diagram of ABBA Protocol used in Alea-BFT

2.4.3.C Common coin

Alea-BFT realises the common coin required by the ABBA primitive using threshold signatures using the protocol proposed by Cachin et al. [29].

In this protocol, every (honest) node signs a unique deterministically chosen bit string – the "name" of the coin – and broadcasts their signature share to other nodes. Afterwards, nodes reconstruct the full signature from the received coin signature shares and obtain the coin's value by passing the signature through an unpredictable function that maps it into a single bit (0 or 1).

Aside from producing coin shares (threshold signature shares) and combining them, nodes must disseminate their coin shares during each ABBA round at the time of coin sampling. To this end, each node broadcasts an additional `COIN` message containing their share of the coin.

2.4.3.D Broadcast Component and Queues

The broadcast component of Alea-BFT manages each node's queue, ensuring that queues are replicated throughout the system and adding new batches of transactions to each node's queue. Every queue position (slot) is associated with an instance of the VCB sub-protocol and uniquely identified by the tuple $<q, s>$, where q is the queue/node identifier, and s the slot index in the queue. We will now describe the two responsibilities of the broadcast component: queue replication and appending new batches to each node's queue.

Firstly, to replicate queues from remote nodes, each node listens to VCB protocol messages tagged with $<q, s>$ and forwards them to the associated VCB instance. Then, when VCB instance $<q, s>$ delivers a batch of transactions, it is added to queue q in position s.

Secondly, nodes construct new batches to add to their respective queue. Each node i waits for a fixed number (the batch size) of new transactions and forms a new batch b_s^i for slot s_i in its queue, starting from slot 0. Afterwards, the node inputs b_s^i to the VCB instance $<i, s_i>$ – which it leads – adding b_s^i to queue i on all (correct) nodes. Finally, the node increments s and repeats this process. For simplicity, nodes receive batches for their own queues by following the VCB instances that they lead.

In summary, this component extends the VCB primitive to N ordered sets/queues of batches. It allows every node to observe a consistent albeit possibly incomplete log of proposals for each node and ensures that correct nodes eventually receive each other's proposals.

2.4.3.E Agreement Component

The agreement component of Alea-BFT decides the delivery of batches of transactions. In every agreement round, a previously agreed-upon queue selection policy chooses a node queue, and an instance of ABBA decides whether to deliver the head of the said queue. Despite ABBA being a leaderless protocol, we refer to the owner of the chosen queue as the agreement round leader. The authors chose a round-robin queue selection policy for their experiments, but they note that different policies are possible. The protocol proceeds as follows for each round r.

Firstly, if a node has received the batch corresponding to an agreement round, it inputs 1 to ABBA instance r. Otherwise, if it has not yet received the batch (due to network slowdowns or a Byzantine proposer), it inputs 0 to ABBA.

Secondly, if ABBA-r delivered 1, the node delivers the corresponding batch, removing it from the queue, and does nothing otherwise.

Thirdly, there is a crucial detail to consider when delivering batches: ABBA can decide to deliver a proposal that a node will never receive – remember that the broadcast component only guarantees dissemination of proposals from correct nodes, and note that nothing stops a Byzantine node from getting its proposal ordered in the agreement component. If the batch to be delivered is not present, nodes request it from others by sending a special `FILL-GAP` message to all nodes that have input 1 to ABBA, containing the queue ID and the slot number of the missing batch. Any correct node that receives a `<FILL-GAP,` i, s`>` message replies with a `FILLER` message containing the batch <i, s>, if it has received it.

In summary, the agreement component decides the delivery of proposed batches using a simple binary consensus primitive and assures recovery from proposer faults.

2.5 Filecoin Hierarchical Consensus

Filecoin is a permissionless blockchain that aims to provide a fair, transparent, and decentralised data storage market. Similarly to other public blockchains such as Bitcoin and Ethereum, Filecoin's scalability is hindered by consensus [2], which is limited by the existence of (temporary) forks in the blockchain, requiring more time for nodes to converge [38]. In contrast, traditional BFT consensus performs well but hasn't seen use in permissionless blockchains due to its susceptibility to Sybil attacks [39].

To bridge this gap in performance, de la Rocha et al. proposed a hybrid approach: the Filecoin Hierarchical Consensus (FHC) framework [2]. Rather than aggregating all transactions across the network and running consensus over them, the authors propose a hierarchy of networks, where *subnets* (child networks) can operate independently from their parent. Additionally, *subnets* are not bound by the rules of the parent chain and can use any consensus protocol, including traditional BFT consensus.

However, partitioning a permissionless blockchain raises security concerns. If transactions are trivially split among network partitions, honest users divide their resources accordingly across all network partitions. By focusing their resources on a specific network network partition, an attacker can overwhelm the partition, performing what is known as a 1% attack. In the context of FHC, this problem is sidestepped by isolating *subnets* from each other: users must move their on-chain resources to a *subnet* to conduct transactions on that *subnet*, explicitly choosing to participate in it. Additionally, *subnets* are regularly checkpointed to their parent network, adding the security guarantees of the parent network on top of their own.

When choosing a real-world system to integrate Alea-BFT into, Filecoin emerged as the winning pick due to the FHC initiative. Although FHC has not yet seen widespread use, it is under active development from its creators and does not constrain our implementation, allowing for future code reuse and flexibility.

3 Implementing Alea-BFT

Contents

In this chapter, we visit all aspects about our implementation of Alea-BFT, excluding optimisations, which we present in the Chapter 4. We begin with an evaluation of various strategies considered for implementing Alea-BFT. Afterwards, we present the framework upon which our Alea-BFT implementation stands, summarise the various pre-existing components for said framework, and introduce the additional supporting components that Alea-BFT uses. Finally, we describe the implementation of Alea-BFT's sub-protocols (VCB and ABBA), components (broadcast and agreement), and the new ”orchestrator” component which coordinates said components.

3.1 Plan of Action

Our work aims to build and evaluate a production-ready implementation of Alea-BFT, and take steps towards its integration in real-world systems. To this end, we had an initial set of desires for this new implementation:

- **Modularity and Testability** Alea-BFT sub-protocols must be independent of the remaining components, allowing for independent testing.
- **Performance** The overhead imposed by the implementation should be minimal. Protocol latency should be dominated by the network and threshold cryptography.
- **Constant Memory Usage** The original Alea-BFT description does not cover how old protocol messages are garbage collected for simplicity. We wish to limit maximum memory usage to a predefined constant, by restricting sub-protocol instantiation with a sliding window (see Section 3.4.2).
- **Rust** The Rust programming language [40,41] is known for its excellent performance and reliability, and is seeing growing usage in the context of blockchain technology. We believe it is a good fit for Alea-BFT.
- **Real-World Applicability** The implementation should be easy to integrate into FHC – a real-world application for Alea-BFT.

Apart from Alea-specific concerns, distributed systems require various supporting components such as a network stack for node communication. Despite FHC not imposing any particular requirement on a *subnet*'s consensus protocol, it can be difficult to align Alea-BFT with Filecoin's consensus, network and other interfaces in practice. For this reason, we chose to first investigate how to integrate a new consensus protocol into FHC, and use the resulting information to guide our implementation decisions.

In the next section, we summarise the results of this investigation.

3.2 Integrating a New Consensus Protocol into FHC

To integrate a new consensus protocol into FHC, the Filecoin clients must be updated accordingly to instantiate and use it. Lotus, one of the official Filecoin clients, was modified in an experimental version to support FHC by Protocol Labs (the company behind Filecoin). Therefore, we focus our work on integrating with this variant of Lotus specifically.

Despite Lotus being written in Go, using Rust was not initially discarded, as it has excellent support for interfacing with other languages, and it is also possible to create a stub consensus module that delegates all work to another process in the same machine, which can be coded in any language, namely in Rust.

We have identified three strategies for integration which we will now describe, ending on the chosen strategy.

3.2.1 Direct Integration in Lotus

Firstly, it is possible to integrate with Lotus directly, by adapting Alea-BFT to work with its consensus interface:

```
interface Consensus {
    ValidateBlock(Context, FullBlock) (error)
    ValidateBlockPubsub(Context, bool, Message) (ValidationResult, error)
    IsEpochBeyondCurrentMax(ChainEpoch) bool
    Type() ConsensusType
    CreateBlock(Context, Wallet, BlockTemplate) (FullBlock, error)
}
```

Listing 3.1: Lotus Modular Consensus Interface [8]

This strategy allows for reusing the validation logic from existing implementations, leaving only open the question of how Alea-BFT interacts with the `CreateBlock` method, which proposes a new block for consensus.

However, it was not trivial to construct a Filecoin hierarchical consensus protocol from Alea-BFT, as it was not immediately clear which properties (e.g., signatures) are required by Lotus for all (sub)chains, or if some are specific to the main chain consensus protocol. Furthermore, this interface does not include any mechanism to pass the stream of ordered blocks from the consensus protocol back to Lotus.

Since existing FHC integrations were already available in the Lotus code base, this strategy was abandoned in favour of reusing existing integrations.

3.2.2 Tendermint Interface

Tendermint [14] is a permissioned blockchain that uses a classic BFT SMR protocol. It was the first permissioned blockchain to be integrated into Filecoin and its Lotus client. The authors of FHC opted to reuse the existing production-ready Tendermint implementation and bridge it to the Lotus client through its Remote Procedure Call (RPC) interface. Under this strategy, Tendermint would run in its own process, co-located with Lotus. Lotus would submit blocks to the co-located Tendermint process and process the stream of ordered blocks from Tendermint using the two existing RPC endpoints.

To integrate Alea-BFT into Lotus/Filecoin, we could completely reuse the existing Tendermint integration, changing the instantiation code to execute Alea-BFT instead. Our implementation of Alea-BFT would implement the Tendermint interface for block submission and ordered block streaming, allowing it to work with the existing integration with little to no changes.

Aside from nearly eliminating all integration work on Lotus's side, this approach would allow us to implement Alea-BFT in any language without adding complexity by leveraging the RPC endpoints as well-defined abstraction boundaries.

Although we were initially interested in this approach, a new FHC integration emerged shortly after (see Section 3.2.3), which led us to abandon the Tendermint approach. In hindsight, this was the right decision since the Tendermint integration was abandoned and removed.

3.2.3 Trantor Modular SMR

After the success of the Tendermint integration, the Lotus developers experimented with integrating another permissioned consensus algorithm – ISS [6]. Interestingly, ISS was implemented on Trantor [7], a modular SMR system, which is itself implemented on Mir [5], a framework for building distributed systems protocols. Rather than integrating ISS directly into Lotus, the authors focused on integrating Trantor itself, allowing future consensus protocols implemented on top of it to be easily included in Lotus.

Mir/Trantor is modular, making it easy to embed in other systems, as long as they are written in Go (like Lotus), and to facilitate tight integration to existing components – Trantor and Lotus share the same network stack and cryptography. Although it is not yet production-ready nor thoroughly optimised, Trantor boasts impressive performance: it attained a throughput of over 30k tx/s with 32 replicas distributed across five continents.

With this in mind, we decided to implement Alea-BFT on top of Mir/Trantor, as it made integration in Filecoin trivial and removed the burden of implementing several components, such as the network stack and cryptographic signatures. However, this approach made us exclude the usage of Rust to implement Alea-BFT, as the efforts it required (inter-process communication or in-process calls using a C-compatible interface) negated the advantages of component reutilisation. We describe the Mir framework and Trantor's components in detail in the next section.

3.3 The Mir Framework and Trantor

The Mir framework [5] enables fast development of distributed protocols by providing an event-driven execution engine for an arbitrary set of modules, supporting tracing, debugging and fault-injection. Concretely, it allows events to be intercepted for recording, replaying or modification before continuing to their destination module. The capabilities of Mir are fully realised in Trantor, a modular SMR system that was developed alongside Mir and includes components providing common services used in distributed protocols such as communication, storage, and cryptography.

In this section, we describe not only the architecture of Mir but also the architecture of Trantor [7], which provides a set of Mir modules with well-defined abstractions that together form an SMR system.

3.3.1 Overview

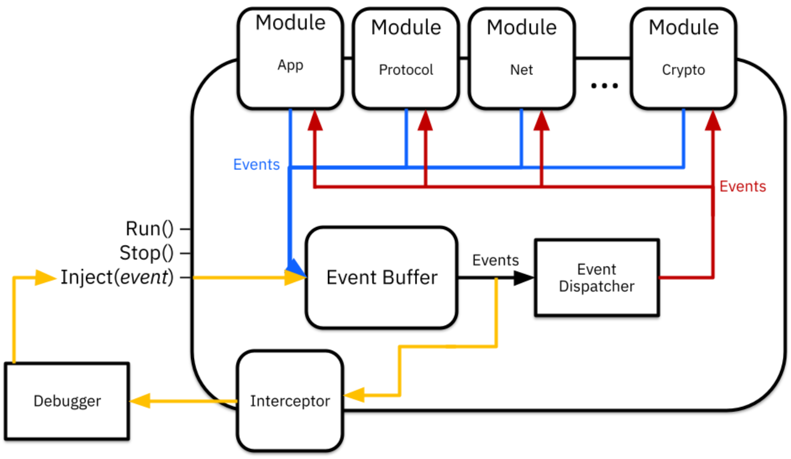


Figure 3.1: Mir Framework High Level Architecture [5]

The Mir framework was designed to implement distributed protocols and follows the conventional distributed systems model, representing a system as a set of communicating nodes. Each node is an independent Mir instance capable of communicating with the remaining nodes through messages over a network. Inside each Mir node is an event loop (represented in Figure 3.1), continuously routing events to and from local modules, which concurrently execute portions of the distributed protocol (or sub-protocols). The event loop dedicates a Goroutine per module to forwarding events from the global event buffer to the Goroutine's corresponding module.

To aid debugging, Mir allows events to be intercepted before dispatch, allowing a debugger to record a full trace or collect metrics of the current execution. Additionally, a debugger can inject events in a node, which allows it to step (deterministic) modules through execution states.

Although Mir only accepts a static set of modules, the runtime extracts a prefix of the destination module identifier (up to the first module identifier path separator). This convention facilitates the existence of module hierarchies and can enable dynamic module management.

3.3.2 Mir Modules

A Mir node is a set of modules that interact with each other through events, and where most modules only generate events in response to incoming events – called **Passive Modules** in Mir. This property greatly simplifies development as the interface of a passive module can be a single event application function (see Listing 3.2) that produces an outgoing event list in response to incoming events. The Mir event buffer receives outgoing events from a passive module through its dedicated event submission Goroutine.

```
interface PassiveModule {
    ApplyEvents(EventList) (EventList, error)
}
```

Listing 3.2: Mir Passive Module Interface [5]

In contrast, modules that interact with the outside world such as timers (which interact with the clock) and the network module must be able to generate events at any time. Instead of an event application function, **Active Modules** expose an event submission function (`ApplyEvents`) and an outgoing event channel (returned by `EventsOut`) as seen in Listing 3.3. Unlike passive modules, active modules require an additional Goroutine dedicated to forwarding events from their output channel to the Mir event buffer.

```
interface ActiveModule {
    ApplyEvents(EventList) (error)
    EventsOut() (chan EventList)
}
```

Listing 3.3: Mir Active Module Interface [5]

By moving non-determinism and interactions with the world to active modules, passive modules can be modelled as fully deterministic state machines, allowing developers to test passive modules or reproduce bugs in simulated environments.

3.3.3 Dynamic Module Management (Factory Module)

In Mir, the root set of modules is fixed at the time of instantiation, but the runtime supports the notion of sub-modules, allowing for dynamic module management. Mir/Trantor include a module designed for dynamic module management named Factory Module, which exposes an interface for explicitly creating instances of a particular module (e.g., the PBFT protocol in the ordering component of ISS-PBFT). Each instance of a sub-module is identified by a number.

The Factory Module also exposes an explicit mechanism for sub-module deletion. The sub-module identifier is also its retention index, and the Factory Module's garbage collection method allows a controller module to remove all modules up to a given retention index.

Although dynamic module management allows for modular code, variance in sub-module instantiation time across nodes may cause severe performance degradation. If a sub-module in a node sends messages to its counterpart in another node before the remote node instantiates it, messages are dropped and then delayed by the nature of re-transmissions. To alleviate this issue, Factory Module instances buffer incoming messages for sub-modules that are not instantiated and are ahead of the current retention index (i.e., were not garbage collected).

However, buffering presents additional challenges, as buffers must be partitioned in order to avoid one single node filling the hypothetical single buffer with (potentially bogus) messages. Additionally, choosing the size of a buffer is a problem on its own, as different modules have different communication characteristics. Furthermore, modules can fully eliminate redundant message information by merging message data into their state, by leveraging module-specific communication properties. We present a different approach to dynamic module management in Section 3.4.2, which drops buffering in favour of automatic module instantiation and sidesteps these issues.

3.3.4 DSL Modules and Code Generation Facilities

Creating a Mir Module can be a tedious process, as it requires writing long multiplexing switch statements that peel away the hierarchy of event types and intricate state management to retain context in request-reply-style inter-module communication (e.g. to request the hasher to hash a value and receive the hash back). Additionally, Mir authors wished to express modules in terms of upon rules, not only to facilitate implementing an algorithm but to ease the analysis of implementation correctness.

To ease module creation and enhance expressiveness, Mir authors built the Domain-Specific Language (DSL) module – a passive module with configurable logic and context storage – and facilities for generating specialised upon rule builders and event emitters from event definitions for use with DSL modules. To facilitate request-reply-style communication, the corresponding events include an origin field that refers to a stored context object in the requester DSL module instance. This field is used by event emitters and upon rule builders to store/retrieve arbitrary context objects for events marked as requests and replies, respectively. Furthermore, DSL modules accept special batched state update handlers, which the module triggers after processing a complete batch of events.

3.3.5 Trantor SMR System

Trantor [7] is a practical modular SMR system realised as a set of modules in Mir, encompassing not only the tasks strictly needed for replication but also checkpointing, garbage collection and online system reconfiguration. Although it currently implements a protocol similar to ISS-PBFT [6] (everything apart from transaction deduplication), Trantor modules can be switched out for replacements from other consensus or TOB protocols as long as they conform to the modules' event-based interfaces.

Applications that want to leverage Trantor only need a corresponding Mir module supporting the interaction in Figure 3.2. Trantor restores the application's state from the checkpoint when it is too far behind the known most recent state and constantly provides an ordered stream of batches of transactions. Occasionally (between epochs), Trantor requests new configurations from the application to apply and a state snapshot to create a checkpoint.

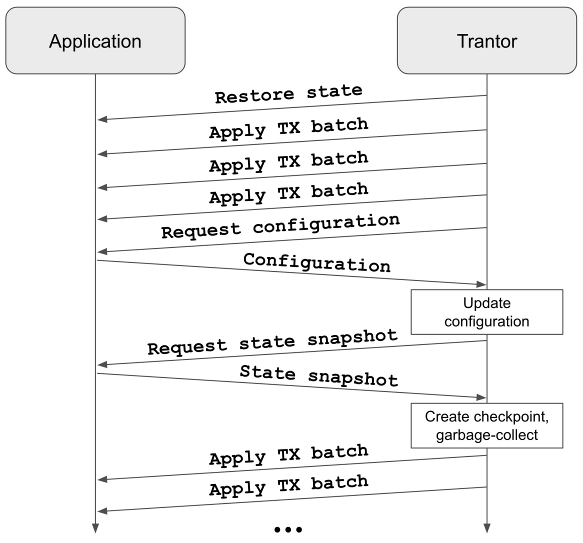


Figure 3.2: Trantor Application Interaction [7]

3.3.5.A Architecture

We will now describe the architecture of Trantor, referencing the diagram in Figure 3.3, which presents all Trantor Mir modules and their connections. The four modules in the lower section of the diagram (Networking, Timer, Hashing and Crypto) are used extensively by all others, and we omit any connections to them for simplicity. Modules highlighted with red (Application, Proposal Validator) are meant to be implemented by the SMR system leveraging Trantor with their custom logic. Modules highlighted with green (Availability, Ordering, Orchestrator) are those we expect to be customised for a specific TOB protocol, and thus our focal point when implementing Alea-BFT.

Trantor splits SMR into five logical stages (numbered in the diagram):

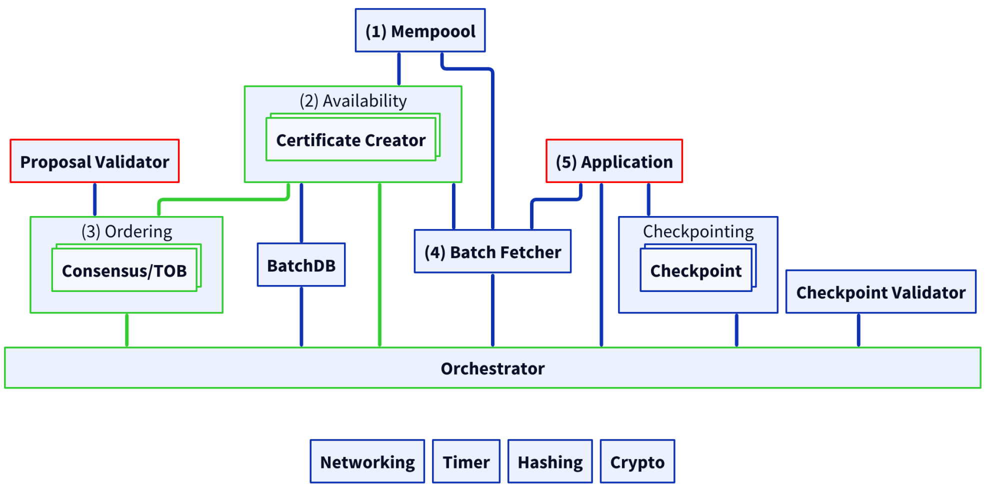


Figure 3.3: Trantor Architecture (based on original architecture diagram from [7])

1. **Mempool**, which aggregates client transactions into batches.
2. **Availability**, which disseminates batches across the system and produces proofs certifying batches are present in enough replicas to ensure the batch is available for fetching from an honest node.
3. **Ordering**, which is a TOB protocol, responsible for ordering dissemination proofs (from the previous stage).
4. **Fetching**, which, given ordered dissemination proofs, fetches the corresponding batches of transactions from local node storage or other nodes if they are missing (e.g., due to a faulty proposer). The Batch Fetcher module additionally filters out duplicate transactions based on a client identifier and a client-controlled counter (transaction number) and serialises delivery of new batches with snapshot requests for checkpointing.
5. **Execution**, which corresponds to the business logic specific to a particular SMR system (implemented by the Application module).

However, these five stages are not sufficient to realise SMR in practice, and several supporting modules are required:

- **BatchDB** stores transactions durably.[1]
- **Checkpointing** creates checkpoints of system/protocol/application state.

[1] Currently, BatchDB only has an in-memory implementation.

- **Proposal Validator** validates proposals from other nodes with application-specific logic. It is optional and not implemented within Trantor (interface only).
- **Checkpoint Validator** validates checkpoints from other nodes. The choice of checkpoint validator is dictated by the choice of checkpointing implementation.
- **Networking** sends/receives messages to/from other nodes over the network.
- **Timer** allows other modules to emit events on a fixed interval or after a timeout.
- **Hashing** hashes arbitrary data.
- **Crypto** generates and validates digital signatures.

Crucially, checkpoints make garbage collection of old sub-protocol instances possible. Since checkpointing and garbage collection involve most components, a special Orchestrator module coordinates most interactions between modules, ensuring operations are adequately synchronised. The Orchestrator module in Trantor manages the transaction log, a sequential list of transactions that captures the state transitions for the application state machine. It is equivalent to the state machine since it can reconstruct the application state by applying all its transactions in order. This log is identical across all nodes except for entries at the end of the log that may not yet be present on some nodes.

Practical applications of SMR must be bound in memory usage, meaning the transaction log must not grow indefinitely. To enable garbage collection of the transaction log, the Orchestrator occasionally checkpoints the application state and replaces a prefix of the transaction log with the checkpoint. It additionally instructs other Trantor components, such as the BatchDB, to delete data related to old transactions.

In Trantor, the transaction log is divided into epochs, and an epoch length is a certain number of transactions fixed at the beginning of the epoch. After the transaction log accumulates enough transactions, the current epoch ends and a checkpoint process begins. During this process, the Orchestrator requests a state snapshot and a new system configuration from the application, which are identical across all honest nodes. The resulting checkpoint certificate allows any node in the system to fast-forward to the valid state captured in the checkpoint.

However, removing old transactions presents a problem for nodes lagging behind the rest of the system, as they might get stuck in an old state. The Orchestrator module also oversees the recovery process for nodes that lag behind the rest of the system. It keeps track of other nodes and dispatches the latest checkpoint to those that fall behind. It also receives and applies incoming checkpoints that refer to a more recent version of the transaction log, coordinating the process with other Trantor components.

3.3.5.B Integration with other systems

Trantor's loosely coupled modular architecture makes it easy to embed in other systems. Each module has a well-defined interface, allowing it to be swapped for a compatible implementation.

In the case of Filecoin/Lotus, Trantor is configured with adapters for Lotus' mempool, networking stack and cryptography stack.

In the case of Alea-BFT, our attention is focused on the Ordering and Availability modules, which are akin to Alea-BFT's agreement and broadcast components, respectively. Since these components have tighter coupling in Alea-BFT than in ISS, we also built a separate Orchestrator module better adapted to our needs. Alea-BFT also requires additional supporting components, such as threshold cryptography, which we describe in the next section.

3.4 New Supporting Modules

While Trantor already includes a network stack and a cryptography stack, it lacks support for threshold cryptography. Furthermore, we considered some of the existing abstractions too limiting for real-world usage, particularly those related to dynamic module management.

In this section, we present the new supporting modules added to Trantor and describe the additions to existing Trantor modules that made the integration of Alea-BFT possible, of which we highlight a more efficient abstraction for dynamic module management that we developed to replace the existing Factory Module – the Modring.

3.4.1 Threshold Signatures

Alea-BFT requires threshold signatures to realise its common coin and the VCB sub-protocol. Similarly to the exiting `crypto` module, we developed a `threshcrypto` that adapts any threshold signature backend conforming to a standard Go interface into a Mir module with an event-based interface. We begin by presenting the new interface for threshold signature backends, then briefly describe the `threshcrypto` module, summarise the various developed implementations for this interface, and finish by introducing a reusable primitive for collecting signature shares and computing the full signature in other Mir modules.

The threshold signature backend interface expected by the `threshcrypto` module (Listing 3.4) is identical to its counterpart for digital signature in the `crypto` module as both contain methods for creating and validating signatures. However, threshold cryptography splits the private key into shares, only permitting the creation of signature shares. Therefore, the interface for threshold cryptography accordingly exposes a method for creating signature shares (`SignShare`) rather than (full) signatures, and verification methods for both kinds of signature (`VerifyShare` and `VerifyFull`). Additionally, threshold cryptography requires the ability to recover the full signature from shares, for which the `Recover` method, not present for regular cryptography, is used. Furthermore, we opted to represent signature shares and full signatures with distinct types rather than regular byte slices to avoid confusion between full signatures, signature shares, and data to be signed.

```
type SigShare []byte
type FullSig []byte

interface ThreshCrypto {
    // Creates a signature share for the provided data using the current node's
    // private key.
    SignShare(data [][]byte) (SigShare, error)

    // Validates a signature share for the provided data against a group member's
    // public key.
    VerifyShare(data [][]byte, signatureShare SigShare, nodeID NodeID) error

    // Guaranteed to succeed, recovering a full signature, when signatureShares
    // contains at least <threshold> signature shares created with SignShare for
    // the same data by different nodes.
    Recover(data [][]byte, signatureShares []SigShare) (FullSig, error)

    // Validates a full signature for the provided data against the group public key.
    VerifyFull(data [][]byte, signature FullSig) error
}
```

Listing 3.4: Threshold Cryptography Interface

The `threshcrypto` module itself is also identical to the `crypto` module. It defines two events for each interface method: one for requesting the cryptographic operation, containing the corresponding method arguments, and another to deliver the result of the operation, containing the corresponding method return value(s). All value conversions between event and interface argument/return types are trivial apart from the `data` argument: this value is hashed using a cryptographic hash function (SHA256) before being passed to the `crypto/threshcrypto` backend. This transformation is not necessary but increases consistency in the duration of cryptography operations by compressing the data to be signed into a constant-size digest.

Finally, we developed three different backends for the `threshcrypto` module: `dummy`, `tbls` and `tbls-herumi`. All backends are initialised with the threshold of nodes required to compute a full signature, and, similarly to the `crypto` backends, may be instantiated with deterministic pseudorandom keys. This deterministic instantiation enables distributed test deployments without the challenges of key distribution. Regarding the backends themselves, the `dummy` backend is meant to accelerate tests and performs no actual cryptography: it returns a single fixed signature/signature share and only accepts this fixed value as valid for all data.

The remaining two backends – `tbls-kyber` and `tbls-herumi` – implement BLS12-381 threshold signatures. While both BLS backends are adequate for our use case (and even rely on the same cryptographic construction), performance problems initially attributed to cryptographic operations prompted us to replace the `tbls-kyber` backend, a Go library [42] used in a real-world system, that appeared to introduce allocation-related overhead. Its tentative replacement – `tbls-herumi` – uses a C/Assembly-based library [43] and therefore does not stress the Go garbage collector. Despite the difference in performance being minor, we valued the consistent duration of cryptographic operations and decided to use `tbls-herumi` as our main backend.

The `threshcrypto` module, including the `dummy` and `tbls` backends, was already contributed back to the original Mir/Trantor code base [44].

3.4.1.A Threshold Signature Aggregator

Constructing threshold signatures is significantly more complex than regular digital signatures, as they require collecting a set of signature shares, and only then combine them into the full signature with `Recover`. Additionally, the presence of Byzantine nodes complicates the recovery process, introducing share verification requirements and potentially requiring multiple calls to `Recover`. Furthermore, the process lends itself to optimisation, particularly regarding signature share verification. To abstract away threshold signature creation and its relevant optimisations, we created a reusable construct that performs this task and can be applied to any Mir DSL module (Section 3.3.4).

The initialisation of the threshold signature aggregator requires two parameters: the threshold of nodes required to compute a full signature and a function that fetches the data to be signed. This function can temporarily return `nil` until the data to be signed is ready. However, it must eventually return (non-`nil`) data to be signed and must return the same data in subsequent calls.

The aggregator exposes two operations: `Add` and `FullSig`. A user of the aggregator calls `Add` repeatedly to input signature shares into the aggregator, along with the ID of the node that produced each share. Eventually, enough shares will be input, the signature is reconstructed, and then it can be retrieved with `FullSig`. Before the signature reconstruction is finished, `FullSig` returns `nil`. Since the creation of signature shares is already accomplished with a single operation, and their dissemination varies widely between use cases, we explicitly leave these steps out of the aggregator's scope.

The exact details of the algorithm for threshold signature reconstruction are presented in Section 4.4, along with its relevant optimisations.

3.4.2 Modring

Dynamic module instantiation in Mir relies on special modules that instantiate and manage other (sub-)modules, where the parent must explicitly forward events to its children. While the Factory Module included in Mir already allows for dynamic module instantiation, it relies on buffering incoming messages for sub-module instances that were not yet created. However, due to its generic nature, Factory Module instances cannot avoid buffering redundant messages, as it is oblivious to the communication properties of the instances they manage. Furthermore, they buffer messages for any sub-module ahead of some retention index, despite many applications (e.g., Alea-BFT's Broadcast and Agreement components, or ISS's orderer factory module) only being interested in a small contiguous subset where each node expects the system to be operating.

To address these challenges, we focused on applications with a small contiguous set of sub-module instances of interest, and devised a new abstraction for dynamic module management based on a fixed-sized sliding window, which we call Modring. Rather than buffering messages or events for not-yet-created sub-module instances, Modrings instantiate sub-modules as needed or drop messages/events if the destination instance is too far ahead of the current window. This approach also facilitates controlling resource usage in the node and tuning tolerance to network delays. By sizing the window larger than required, we expect nodes to remain reasonably synchronised in the presence of latency fluctuations at the cost of increased resource utilisation. Concretely, we expect sub-module instances retained in the window's first slots to help slow nodes catch up and the instances in the last slots to accumulate messages from faster nodes.

Additionally, the Modring exposes methods for detecting and recovering nodes lagging too far behind the rest of the system, particularly nodes stuck in windows of sub-module instances that can no longer make progress (because they were already freed in other up-to-date nodes). To detect stale nodes, Modring exposes incoming messages directed at freed sub-module instances, allowing up-to-date nodes to notice attempts to make progress in freed instances. To recover stale nodes (e.g., after the stale node receives a checkpoint message), Modring allows nodes to force their window to move to an arbitrarily distant future state.

In the remainder of this section, we present the lifecycle of sub-module instances within a Modring and how it interacts with the operating window, followed by Modring's control interface.

3.4.2.A Sub-module Instance Lifecycle and Operating Window

In a Modring, sub-module instances are automatically created but manually freed. However, sub-module instantiation is limited by a sliding window called the operating window – only sub-module instances inside the operating window can be created. This window has size `Size`, starts in position 0, and automatically advances as sub-module instances are freed. We will now describe in detail the lifecycle of a sub-module instance within Modring, referencing the state transition diagram in Figure 3.4.

Initially, nearly every sub-module instance is ahead of the current operating window – the ”Future” state – where events directed to the instance are dropped. When a sub-module instance enters the op-

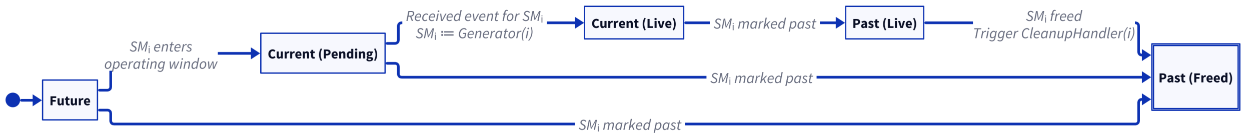


Figure 3.4: Modring sub-module instance i (SM_i) State Transition Diagram

erating window – the ”Current” state – the Modring begins forwarding events directed at it. Eventually, ”Current” sub-module instances are marked as past, leaving the current operating window and transitioning to a ”Past” state. In this situation, Modring ceases forwarding events to them (except for ”Past (Live)” sub-module instances, as we describe in the next section).

However, sub-module instances are lazily created and freed, which results in ”Current” and ”Past” states having two variants: ”Live” and not live (”Current (Pending)” and ”Past (Freed)”). ”Current” module instances start in the ”Current (Pending)” state and transition to the ”Current (Live)” state when they are initialised. This initialisation is triggered by the first event directed at the sub-module instance while in the ”Current (Pending)” state, which causes the Modring to create the new instance using the `Generator`. Regarding freeing of sub-module instances, ”Current (Live)” instances initially transition to the ”Past (Live)” when marked past and are only freed (transition to ”Past (Freed)”) once a ”Current (Pending)” instance being created requires them to. Concretely, Modring stores live sub-module instance metadata in a ring buffer and defers freeing ”Past (Live)” instances to the moment when their slot in the ring buffer is required for another (”Current (Pending)”) instance. Note that sub-module instances may be marked as ”Past” while other instances behind them are still in ”Current” or ”Future” states, forcing their metadata to be stored in the ring buffer and potentially freeing ”Past (Live)” instances in their slots.

3.4.2.B Modring Control Interface

Unlike the Factory Module, which is controlled through Mir events for sub-module instance creation and freeing, Modring exposes an interface based on regular function calls. We designed this interface to be used from an arbitrary `controller` module, which interacts with the rest of the system using regular Mir events and can use any logic to control the Modring. To facilitate this pattern of partitioning controller and children modules, we created a new abstraction – RoutedModule – which forwards incoming events to a `controller` module in case they are directed at the root, or to another module (in this case the Modring) in case they are directed to the children. To avoid complexity in sub-module instance lifecycle management and synchronising Modring internal structures, all `controller` events are processed before children events.

A Modring is initialised with the following parameters:

- `Size` – the maximum number of concurrent sub-module instances at any given time.
- `Generator` – a function that creates a new sub-module instance given its identifier, and a list of events to be emitted at the time of creation.
- `CleanupHandler` – a function that is called whenever a sub-module instance is freed. It accepts the sub-module instance identifier, and returns a list of events to be emitted by the Modring.
- `PastMessageHandler` – a function that is called when messages are received for sub-module instances that are behind the current working window. It returns a list of events to be emitted by the Modring.
- `ModuleID` – the root of the sub-module hierarchy managed by the Modring.

The operations exposed by the Modring are as follows:

- `AdvanceViewToAtLeastSubmodule(p) -> ()` – Marks all modules up to position $p-1$ as Past, leaving the operating window starting in sub-module instance p.
- `FreePast() -> EventList` – Frees all Past sub-modules that are still live. Returns events generated by `CleanupHandler` for each freed sub-module instance.
- `MarkSubmodulePast(p) -> EventList` – Marks sub-module instance at position p as past. This may free a live instance p' in a position previously marked as past. In that scenario, this operation returns the events generated by `CleanupHandler` for instance p'
- `IsInView(p) -> {True, False}` – Checks if sub-module instance p is inside the operating window.
- `IsInExtendedView(p) -> {True, False}` – Checks if sub-module instance p is live or inside the operating window.
- `MarkPastAndFreeAll() -> EventList` – Marks all sub-module instances as past, and frees all live instances (returning their respective `CleanupHandler` events).

3.4.3 Snooze Net

Mir/Trantor includes a network stack based on Transport Control Protocol (TCP) (TLS/Noise if the network is untrusted) and a variant of stubborn channels [45]. Together, they guarantee message integrity and eventual delivery, albeit with the possibility of message duplication. However, it requires that all modules buffer or process all incoming messages to guarantee eventual delivery. This requirement is hard to satisfy in practice (node resources are limited) and impossible in the context of Modrings, which drop incoming messages outside of their operating window by design. Additionally, the current network stack includes a buffer for outgoing messages, whose size must be carefully chosen to preserve the eventual delivery guarantee: if the buffer is too small, the network stack may drop outgoing messages. To address both of these issues, we developed a new module – Snooze Net[2] – that allows nodes/modules to drop incoming messages, by automatically re-transmitting them until they are (implicitly or explicitly) acknowledged. Concretely, Snooze Net wraps the existing network module with a similar variant of stubborn channels, enqueueing messages separately and re-transmitting them until they are acknowledged, guaranteeing that they are eventually delivered successfully to their destination module.

Snooze Net is initialised with two parameters: the maximum number of messages re-transmitted in one go (`MaxRetransmissionBurst`) and the interval of re-transmission (`RetransmissionLoopInterval`). It exposes an event-based interface, shown in Listing 3.5 as Golang-formatted pseudo-code, which includes events for sending messages, acknowledging messages, and marking messages as received. In particular, sending messages is similar to the `net` module with the addition of a `CustomID`, which uniquely identifies a message when combined with the IDs of the destination module and destination node. The choice of custom message ID is specific to each module and is often discernable from context. Therefore, it is not sent across the network by default and must be passed in all Snooze Net operations.

Regarding resource usage, Snooze Net's re-transmission queue size must be bounded. This is accomplished by limiting the number of destination-ID pairs that can be scheduled for re-transmission. Note that Snooze Net's resource usage in each node is bounded by the maximum number of destination-ID pairs that can be scheduled for re-transmission. By combining Snooze Net with Modrings and protocols that require a limited number of message types (custom IDs), the re-transmission queue size is bounded by design.

In the following sub-sections, we specify how messages are re-transmitted and acknowledged.

3.4.3.A Re-transmission

Snooze Net's re-transmission mechanism differs from protocols like TCP because the underlying channel is already stubborn. Therefore, it focuses on re-transmitting messages that were received but not processed by their destinations (messages left unacknowledged). Additionally, Modring window sizes are expected to be large enough to dampen the effects of network latency spikes, further minimising the need for re-transmissions. Therefore, we expect re-transmissions to be required only when a node lags far behind the rest of the system. Considering this expectation, Snooze Net re-transmits a small fixed number of messages (`MaxRetransmissionBurst`) at a configurable interval (`RetransmissionLoopInterval`), which is set to a value much larger than the expected network latency (e.g., several seconds).

[2]Snooze Net is named Reliable Net in the code base because it was initially developed to circumvent a reliability bug in the Mir network stack.

```
type CustomID string

interface SnoozeNet {
    // Each method corresponds to a Mir Event that Snooze Net can process as described.

    // Sends message msg to all nodes in destinations, queueing it for re-transmission
    // to each destination until they acknowledge it.
    // Note: destination module ID is a property of msg
    SendMessage(msg Message, id CustomID, destinations []NodeID)

    // Sends acknowledgement of message identified by (destModule, id) to sender.
    Ack(destModule ModuleID, id CustomID, sender NodeID)

    // Marks message identified by (destModule, id) as received by nodes.
    MarkRecvd(destModule ModuleID, id CustomID, nodes []NodeID)

    // Marks messages directed to destModule and its sub-modules as received by nodes.
    MarkModuleMsgsRecvd(destModule ModuleID, nodes []NodeID)

    // Executes re-transmission routine.
    // Re-transmits up to MaxRetransmissionBurst messages that were enqueued for
    // re-transmission before the last execution of the re-transmission routine and
    // reschedules the routine for execution if the queue is not empty.
    Retransmit()
}
```

Listing 3.5: Snooze Net Event Interface

We now describe the re-transmission routine, which is only scheduled when required. Concretely, whenever the message queue becomes non-empty and the re-transmission routine is not scheduled for execution, the timer module is configured to signal the Snooze Net module with the `Retransmit` event after `RetransmissionLoopInterval` time. The re-transmission routine (triggered by `Retransmit`) first checks if the queue is empty. If so, all messages were acknowledged, the routine registers that it is no longer scheduled for execution and exits. Otherwise, there are unacknowledged messages, and the routine re-transmits up to `MaxRetransmissionBurst` messages, considering only those that were added to the queue before the last execution of the routine to avoid needless re-transmissions. This operation advances a cursor in the re-transmission queue, which controls the next set of re-transmitted messages, ensuring that all queued messages are eventually re-transmitted. Note that the cursor circles back to the beginning of the queue when it reaches the end of the queue. Finally, the re-transmission routine reschedules itself for execution using the timer module.

Despite the low rate of re-transmissions, the current implementation does not consider network conditions and may completely fill the `net` module outgoing message buffer, preventing other (more recent) messages from being sent. To combat this, Snooze Net could be integrated into the `net` module, which entails refactoring the `net` module to be generic over the underlying transport library (libp2p or gRPC).

Furthermore, re-transmissions have no concept of priority and may occupy the network with messages that are not critical to system progress, inflating its latency. Resolving these issues is a significant undertaking and is left for future work.

3.4.3.B Acknowledging a Message

Snooze Net supports both explicit and implicit message acknowledgements. Explicit acknowledgements are sent using the `Ack` event, which prompts Snooze Net to send an acknowledgement message to the sender node, carrying the unique identifier of the message being acknowledged. Snooze Net handles these explicit acknowledgement messages directly, removing the acknowledged message from the re-transmission queue.

Implicit acknowledgements allow acknowledgements to be piggybacked on existing protocol messages. After a module receives a message that implicitly acknowledges the previously sent message, it can tell Snooze Net to mark messages as acknowledged using `MarkRecvd`. Additionally, all messages with a given destination module prefix can be marked as acknowledged using `MarkModuleMsgsRecvd`, which is used by Modring controllers to clear all messages from a freed module from the re-transmission queue. SnoozeNet indexes queued messages by destination module to ensure `MarkModuleMsgsRecvd` is computationally efficient.

3.4.4 Threshold-Signature-based Checkpointing

Trantor already includes a checkpointing protocol, which collects signatures from a quorum of nodes on a summarised representation of the application state. However, this approach results in different nodes potentially having different views of the checkpoint signers, which Trantor/ISS solves with an additional instance of consensus to decide the set of signatures included in the checkpoint.

To avoid this instance of consensus, we created a variant of Trantor's checkpointing implementation using threshold signatures. Rather than sending a checkpoint signature to all nodes, each node sends a signature share to all other nodes, and the threshold signature is reconstructed from the shares. Since the original protocol required $\lceil \frac{N+F+1}{2} \rceil$ signatures (a Byzantine quorum), we decided to require an equal threshold of $\lceil \frac{N+F+1}{2} \rceil$ shares in the threshold signature cryptosystem. However, future exploration may be able to lower this bound to $F+1$, as all correct nodes hold the same application state.

3.5 Verifiable Consistent Broadcast (VCB)

The VCB protocol used by Alea-BFT's broadcast component (see Section 2.4.3.A) was realised as a Mir module that can be instantiated multiple times and, in particular, supports concurrent instances of VCB.

Since VCB has the leader of the broadcast also act as a follower, the VCB module encapsulates the behaviour of both leader and follower nodes, conditionally enabling leader behaviour when the leader ID matches the current node ID. This module requires three parameters for initialization: the ID of the VCB instance (included in data to be signed), the set of participating nodes, the ID of the leader node, and the ID of the current node. Similarly to other Mir modules, it exposes an event-based interface consisting of the following events:

- `InputValue` (leader-only, inbound) – provides input to the VCB instance. For efficiency reasons, this event accepts both the list of transactions to be broadcast and the list of corresponding transaction IDs. The transaction IDs are computed by the mempool module and are required for computing the VCB threshold signature. However, the mempool already provides the transaction IDs when delivering a new batch, so the VCB module allows the invoker to pass them and avoid redundant work.
- `Deliver` (outbound) – conveys the result of the broadcast to the parent module. Includes the stored batch ID, the threshold signature, and the ID of the VCB module emitting the event.

  Note that the VCB module ID allows the parent module to distinguish between different VCB instances. In the case of VCB, followers deliver a broadcast without any input, rendering the Mir DSL module's request-reply helpers (Section 3.3.4) useless in this situation.
- `QuorumDone` (leader-only, outbound) – signals that $\lceil \frac{N+F+1}{2} \rceil$ nodes indicated they completed the broadcast. In particular, this also implies that at least $F+1$ correct nodes received the broadcast value. In the context of Alea-BFT, VCB leaders must wait for this event before freeing the broadcast instance to ensure the corresponding queue slot to this broadcast is eventually delivered. Additionally, it is used for VCB duration estimates, which is useful for a set of optimisations that we describe in Section 4.3.
- `AllDone` (leader-only, outbound) – signals that all nodes indicated they completed the broadcast. It is used for VCB duration estimates, which, again, are used by our optimisations explained in Section 4.3.

The VCB module closely follows the protocol description, with the following key differences:

- After the `FINAL` message is processed, followers reply to the leader with a new `DONE` message. Furthermore, `FINAL` messages are immediately replied to with `DONE`.

  The leader registers the reception of `DONE` messages to emit `QuorumDone` and `AllDone` events at their appropriate moments.
- Network communication is realised with Snooze Net. Leader `SEND` messages are implicitly acknowledged with `ECHO` messages, which are in turn implicitly acknowledged by the leader's `FINAL`

message. The leader's FINAL message is implicitly acknowledged by the new DONE message, which bypasses Snooze Net and thus does not require an acknowledgement.

- The broadcast value (input to the leader or received via the leader's SEND message) undergoes a pre-processing step before any cryptographic operations are performed, that ensures the batch is stored in the BatchDB and the threshold signature data is computed. This step is detailed in Section 3.5.1.
- The threshold signature reconstruction from shares in the broadcast leader uses the mixin presented in Section 3.4.1.A.

Regarding the storage of the broadcast value, the BatchDB module demands a retention index for each batch, which controls when the batch is deleted. However, Alea-BFT is asynchronous and, even after adding a checkpointing mechanism, requires batches to be retained for an indefinite period until they are delivered. Therefore, the retention index is initially set to infinity by the VCB module and the invoker is responsible for modifying the retention index of the batch to allow the BatchDB to eventually free it.[3]

However, this introduces the possibility of a resource leak under the presence of a Byzantine leader. Concretely, a Byzantine leader may send bogus signatures to a small portion of the nodes, causing the VCB instance to fail and storing a broadcast value that may never delivered. Since the broadcast value is never delivered, the parent module cannot modify its retention index, and the batch is never deleted. The solution to this problem is not trivial. In the context of Alea-BFT, freeing the batch of a failed VCB could lead to a race condition with the recovery mechanism (FILL-GAP/FILLER). Even if the parent module or the VCB instance could modify the retention index to free the batch, the Byzantine leader can follow protocol with the remaining nodes, potentially causing the batch to be deleted before it is delivered on the nodes that received the bogus SEND message. A tentative solution to this problem is introducing reference counting to the BatchDB module, allowing batches to be freed upon cleanup of their corresponding incomplete/failed VCB instance only when not in use by other components. Additionally, this problem suggests that dynamic module instantiation helpers (such as the Modring or the Factory Module) should allow sub-modules to specify custom cleanup actions.

We developed and executed a small simulation-based unit-test for this implementation of VCB to validate it under a multi-node fault-free scenario.

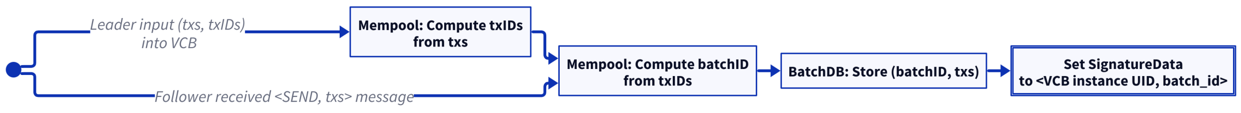
3.5.1 Broadcast Value Pre-Processing

The broadcast value pre-processing mixin is responsible for persisting the batch and computing the VCB threshold signature data. It interacts with the VCB module through two methods: `Input` – which provides

[3]We made a small modification to BatchDB module to allow for the retention index to be modified on stored batches.

the transactions (and, if already computed, their IDs) being broadcast – and `SigData` – which retrieves the data to be threshold-signed. Initially, `SigData` returns `nil` until the pre-processing routine concludes, at which point it returns the data to be signed. This mixin is shared by both leader and follower logic, ensuring that leader nodes (which also act as followers) do not perform the pre-processing routine twice. Furthermore, this mixin prepends a unique string (the VCB instance ID) to the data being threshold-signed, to prevent replay attacks. This instance ID is built by concatenating a system-wide instance ID and the module ID corresponding to the VCB instance.

Figure 3.5 presents an activity diagram detailing the routine, including the conditions upon which `Input` is called. Concretely, it enlists the help of the Mempool module to compute the transaction IDs, then the ID of the batch, and finally asks the BatchDB module to store the batch. After BatchDB stores the batch, the routine computes the threshold signature data and makes it available to the VCB module in its `SigData` method.



Leader input (txs, txIDs) into VCB
Follower received <SEND, txs> message
Mempool: Compute txIDs from txs
Mempool: Compute batchID from txIDs
BatchDB: Store (batchID, txs)
Set SignatureData to <VCB instance UID, batch_id>

Figure 3.5: VCB Broadcast Value Pre-Processing Activity Diagram

3.6 Asynchronous Byzantine Binary Agreement (ABBA)

The ABBA protocol used by Alea-BFT's broadcast component (see Section 2.4.3.B) was realised as a Mir module that can be instantiated multiple times and, in particular, supports concurrent instances of ABBA. This module requires three parameters for initialization: the ID of the ABBA instance (included in the name of the coin), the set of participating nodes and the ID of the current node. Similarly to the VCB module, it uses Snooze Net for network communication with explicit acknowledgement messages and exposes an event-based interface consisting of the following events:

- `InputValue` (inbound) – provides input to the ABBA instance. For efficiency reasons, this event accepts both the list of transactions to be broadcast and the list of corresponding transaction IDs (which are required for a later step and are already computed by the parent).

- `Deliver` (outbound) – conveys the result of the binary agreement to the parent module. Includes the decided bit and the ID of the ABBA module emitting the event.

  Note that the ABBA module ID allows the parent module to distinguish between different ABBA instances. In the case of ABBA, nodes may deliver before providing any input themselves, rendering the Mir DSL module's request-reply helpers (Section 3.3.4) useless in this situation.

- `RoundInputValue` (internal, outbound) – provides input to an ABBA round.
- `RoundDeliver` (internal, inbound) – conveys the output of an ABBA round to the ABBA module.
- `RoundFinishAll` (internal, inbound) – conveys the intent to broadcast `FINISH` (if not broadcast yet) from an ABBA round to the ABBA module.

The ABBA module is responsible for broadcasting and processing `FINISH` messages, emitting `Deliver` events when the binary agreement decides. However, the remaining protocol steps are encapsulated into independent ABBA round sub-modules, managed using a Modring. By allowing multiple ABBA round sub-modules to execute concurrently, we prevent a stall between ABBA rounds where nodes slightly behind the rest of the system miss the initial message of the next round. Additionally, this approach is tunable: by setting a larger window size, the system can tolerate nodes being more rounds behind the rest of the system. Note that, despite allowing concurrent ABBA rounds to coexist, the ABBA protocol was left unmodified. Concretely, round modules do not broadcast any messages nor deliver a result before receiving input. Before receiving input, round modules only register messages received from other nodes (deduplicated, in a constant-size structure).

The ABBA module functions as follows.

1. When ABBA reaches the condition for termination (Byzantine quorum of `FINISH` messages), it forcefully advances the Modring window to free all ABBA round sub-modules. Since `FINISH` messages are enough to ensure protocol delivery and termination, there is no point in continuing to execute ABBA rounds.
2. The main module forwards its input bit to the first ABBA round.
3. The ABBA round i delivers b. The main module frees ABBA round i and inputs b to ABBA round $i+1$. This step is repeated until the protocol reaches the conditions of step 1.

Furthermore, in each ABBA round, the protocol demands the toss of a common coin, which is realised using the protocol presented in Section 2.4.3.C. Concretely, the ABBA round module broadcasts a new `COIN` message containing a threshold signature share of a unique coin name (which is the concatenation of a globally unique system ID, the ABBA instance module ID – unique in each system – and the round number). Each node then reconstructs the full signature using the threshold signature aggregator from the received `COIN` messages and derives from it the value of the coin (a single bit) by hashing it using `SHA-256` and selecting the first bit of the hash.

Later in our work, we realised that most cryptographic hashes do not guarantee unpredictability for each output bit (only for the complete output). Thus, this construction needs to be replaced by a Deterministic Random Bit Generator (DRBG) [46], which can be seeded with the (constant-length) signature hash and used to generate a single truly unpredictable bit. Nevertheless, we do not expect this issue to affect our experimental results because a separate experiment showed that 50.6% out of 3808 coin flips resulted in 1, which is close to the expected 50% of a random coin flip, and the Hash_DRBG algorithm described by Barker et. al [46] only entails a constant number of hash calculations to produce a random bit.

Lastly, to minimise coin-related latency, the coin signature share (local) computation begins when the round sub-module is instantiated. However, the `COIN` message is only sent in the coin toss step to preserve protocol properties.

We developed and executed a small suite of simulation-based unit tests for this implementation of ABBA to validate it under multi-node fault-free scenarios with unanimous and non-unanimous inputs.

3.7 Alea-BFT components

The original description of Alea-BFT divides it into two components: broadcast and agreement. These roughly correspond to Trantor's availability and ordering components, which are responsible for disseminating and ordering client requests, respectively. However, the original design of Alea-BFT does not include a checkpointing mechanism, which is essential for limiting its resource usage by freeing old instances of its sub-protocols. Additionally, applications using Trantor expect to receive ordered batch IDs and obtain the corresponding batches from the `BatchFetcher` module, which in turn expects an availability module to fetch any missing batches. Furthermore, Trantor modules rely on a notion of sequence numbers, which correspond to the position of batches in the log of delivered/ordered batches.

To address these concerns, we refactored Alea-BFT for better integration into Trantor and to include a checkpointing mechanism. Concretely, the failed broadcast recovery mechanism (`FILL-GAP`/`FILLER`) was moved to the broadcast component, the agreement component became a façade for executing agreement rounds in series, and all logic regarding Alea-BFT's queues was moved to a new orchestrator component. This design mirrors Trantor's, where an orchestrator component coordinates the interactions of an availability/broadcast component, responsible for disseminating client requests, and an ordering/agreement component, responsible for ordering them.

Regarding sequence numbers, they can be directly mapped to Alea-BFT's agreement round numbers since each agreement round causes the delivery of at most one batch. Consequently, the sequence number of a batch is the agreement round number that caused its delivery.

Regarding checkpointing, the orchestrator component coordinates their creation and usage. The notion of checkpointing mandated the introduction of epochs to Alea-BFT, which mark the moments when checkpoints are taken. To allow the orchestrator to free resources that are no longer needed, all resources consumed by Alea-BFT are associated with a retention index corresponding to the epoch in which they were last required.

To validate this implementation of Alea-BFT, we adapted ISS's existing integration tests covering various scenarios, namely fault-free, crash fault and censoring a node from all incoming proposals to force the use of the recovery mechanism.

In the rest of this section, we describe the redefined roles of Alea-BFT's components – orchestrator, broadcast and agreement – and their implementation in Mir.

3.7.1 Orchestrator Component

Alea-BFT's orchestrator[4] is responsible for the following duties:

- **Batch Cutting** – Instruct the broadcast component to create and disseminate new batches.
- **Agreement Loop Control** – Provide inputs for agreement rounds to the agreement component. Process the results of agreement rounds, delivering the corresponding slots in Alea-BFT's queues to the application.
- **Checkpointing & Garbage Collection** – Create system checkpoints. Instruct other components to free unneeded resources when checkpoints stabilise. Unlike ISS's checkpointing, this checkpointing process does not block the end/beginning of epochs: the protocol can continue regardless of checkpointing progress.
- **Recovery** – Track other nodes' progress in the system. Send checkpoints to nodes that fell too far behind the rest of the system. Recover a local node using checkpoints received from other nodes.

Despite these duties being orthogonal to each other, there is a non-trivial technical cost associated with creating and coordinating multiple Mir modules. Additionally, if split into different modules, these duties would perform redundant work as they have similar inputs, albeit for different purposes. Therefore, we decided to incorporate them in the orchestrator module and let them share state.

To ease coordination among duties, we developed a Queue Selection Policy construct, that abstracts the state tracking of Alea-BFT's queues and the mapping between agreement rounds and the slots whose delivery is being decided.

We describe the Queue Selection Policy abstraction and orchestrator duties in detail in Appendix A.

3.7.2 Broadcast Component

In Trantor, transaction dissemination is handled by an availability module. This module can create and verify availability certificates, which certify that a given batch was disseminated to at least one correct

[4]In the Alea-BFT Mir code base, the orchestrator component is named director. However, we always refer to it as orchestrator in this document for consistency with the Trantor design document [7].

node (a necessary condition for delivery). Additionally, the availability module must be able to retrieve the batch corresponding to valid availability certificates.

This concept maps well to Alea-BFT's broadcast component, using VCB proofs as availability certificates. However, in Alea-BFT's original description, failed VCB instances are recovered by the agreement component using the `FILL-GAP`/`FILLER` mechanism. To better integrate Alea-BFT into Trantor, we merged this recovery sub-protocol into the broadcast component to form a complete availability module.

Since the set of broadcast queues is static (one per node, and system membership is fixed), we opted not to use the Modring (presented in Section 3.4.2). Instead, the broadcast component is implemented as a custom Mir module, which forwards events to one of the N broadcast queue sub-modules or to a controller module, which is the broadcast component itself but is separate to use Mir's DSL module abstraction (Section 3.3.4). This approach is similar to the RoutedModule pattern used with the Modring and controller module but with a fixed set of sub-modules.

In the remainder of this section, we describe how the broadcast component implements the availability module interface, the broadcast queues, and the lifecycle of transactions in the system concerning epochs.

3.7.2.A Broadcast Component as an Availability Module

Availability modules expose events for requesting and verifying availability certificates and retrieving the batch corresponding to an availability certificate. In Alea-BFT, this availability certificate is a VCB proof and the batch metadata – its slot and content-based hash that identifies it within Mir/Trantor's BatchDB. These events follow a request-reply pattern: `RequestCert`/`NewCert` request and deliver a new availability certificate respectively, `VerifyCert`/`CertVerified` pertain to certificate verification, and `RequestTransactions`/`ProvideTransactions` request and deliver the batch of transactions corresponding to an availability certificate.

Creating availability certificates (`RequestCert`/`NewCert`) boils down to requesting a new batch from the mempool and inputting it into the local node's broadcast queue (in the next unused slot). However, the mempool controls the batch size and does not impose a minimum size (batches can even be empty!), only a maximum size. Thus, we extended the mempool with a minimum batch size parameter that can be set to the same value as the maximum batch size to enforce a fixed batch size, as specified in the original protocol description. Section 4.3 refines this idea by allowing non-full batches to exist during periods of low system load to improve latency.

However, unlike other availability implementations, Alea-BFT may deliver broadcasts originating from any node in the system. Therefore, the request-reply pattern does not fit the creation of new availability certificates in Alea-BFT. Instead, the broadcast component uses a modified version of the `RequestCert` and `DeliverCert` that do not use DSL code generation for request-reply events. Concretely, when any broadcast queue delivers new certificates to the broadcast component, they are forwarded to the orchestrator in modified `DeliverCert` events.

Batch retrieval (`RequestTransactions`/`ProvideTransactions`) is accomplished by fetching the transactions from the BatchDB module using the content-based hash of the batch (present in the availability certificate). However, Alea-BFT reasons only about queue slots, not availability certificates, so we wished to allow the orchestrator to reason only about slots. Therefore, we opted to allow the existence of incomplete availability certificates – containing only the identified slot and store the full availability certificates in the broadcast component indexed by their corresponding slot. This decision also aided in garbage-collecting old transactions, by keeping all relevant metadata in the broadcast component.

Thus, when the batch fetcher module requests transactions for a delivered availability certificate, the provided certificate only identifies a slot in Alea-BFT's queues. To retrieve the transactions, the broadcast component procures the corresponding full availability certificate. If it is present, its transactions are locally available and can be delivered, otherwise, we trigger the `FILL-GAP`/`FILLER` recovery mechanism. When the recovery mechanism finishes, the batch transactions become locally available, the corresponding full availability certificate is stored, and the request for transactions is satisfied.

Availability certificate verification (`VerifyCert`/`CertVerified`) consists of checking the VCB proof of the availability certificate against the batch metadata present in the certificate. Despite being part of the availability module interface, verification is not required by any other module: it is only used in Alea-BFT's broadcast recovery mechanism. Nevertheless, we exposed it for completeness.

3.7.2.B Broadcast Queues

Broadcast queues are implemented with the RoutedModule+Modring pattern presented in Section 3.4.2.B. Their Modring holds all VCB instances related to the queue, which share the same leader – the queue owner.

The queue module abstracts interactions with the underlying VCB instances. When a VCB instance delivers a batch, the queue module generates the corresponding availability certificate and delivers it to the broadcast component. Additionally, if the queue is owned by the local node, it exposes an `InputValue` operation. This operation inputs transactions into the VCB instance with a given slot in the queue.

Additionally, the queue module accepts events for freeing slots in the Modring (allowing the creation of more VCB instances), for informing the queue of new epochs and for garbage-collecting VCB instances associated with old epochs. The lifecycle of VCB instances and their transactions is described in detail in the next section.

3.7.2.C Lifecycle

This implementation of Alea-BFT was designed to be bounded in resource usage while maintaining protocol guarantees. Thus, the broadcast queues must eventually garbage-collect old VCB instances and their respective transactions. To this end, the broadcast component receives and propagates events from the orchestrator to the broadcast queues, informing them of new epochs, requests for garbage collection of old epochs, and restoration from checkpoints.

Broadcast queue slots are freed when the corresponding batch is delivered by the orchestrator to the application. Thus, to garbage-collect old VCB instances, the queues track the last freed slot in each epoch. When instructed to garbage-collect slots for all epochs below a given epoch, the queue module instructs the Modring to free all slots below the last freed slot in that epoch. Additionally, it clears the mapping of epochs to their last freed slot for all epochs below the given epoch. Furthermore, the broadcast component clears all full availability certificates corresponding to slots for old epochs.

However, this mechanism does not garbage-collect old transactions, as VCB stores them with an infinite retention index in BatchDB. To address this, the broadcast component associates the current epoch with a batch of transactions upon their delivery. Concretely, when the batch fetcher requests the transactions corresponding to an availability certificate from the broadcast component, it fetches the transactions and updates the retention index of the batch in BatchDB to the current epoch. Alas, this approach suffers from a problem similar to VCB's in the presence of Byzantine nodes. If a batch is delivered, and an identical batch is later disseminated but only delivered several epochs later, the batch may be garbage-collected and become unavailable in all nodes. This problem can be circumvented by introducing a reference-counting mechanism in BatchDB, similar to the proposal for an identical problem in VCB (see Section 3.5).

3.7.3 Agreement Component

Alea-BFT executes a sequence of binary agreements, deciding in each one whether or not to deliver the head of a broadcast queue. In this implementation, the agreement component is a façade for the serial execution of ABBA instances, delegating the computation/interpretation of ABBA inputs/outputs to the orchestrator.

Unlike the original Alea-BFT description, we allow multiple ABBA instances to exist concurrently to prevent stalls when moving between one agreement round and the next. Concretely, the agreement component is realised with the RoutedModule+Modring pattern presented in Section 3.4.2.B. Despite having multiple ABBA instances, the agreement component still executes a single agreement round at a time. The ABBA instances ahead and behind of the current round passively listen to future agreement rounds and allow message re-transmissions, respectively. To avoid out-of-order deliveries of ABBA outputs, which is possible in this design, we buffer ABBA outputs and deliver them in order.

Modrings have a fixed capacity, so the agreement component frees ABBA instances as they terminate. However, this could render remote nodes stuck in old ABBA instances as more up-to-date nodes garbage-collect them and stop message re-transmission. Thus, the agreement component stores the outputs of ABBA instances as they deliver and, after they are garbage-collected, replies to any message directed at them with a special `FinishMessage`, conveying the decided output for that ABBA instance. The remote node transforms this message into a `FINISH` message and forwards it to the corresponding ABBA instance, allowing it to eventually deliver and terminate. This modification to the `FINISH` message does not compromise ABBA guarantees: the underlying mechanism made possible by the `FINISH` message is similar to PBFT's checkpoint mechanism [13] and unanimity allows a node to determine the final value of ABBA protocol, so unanimity is a sufficient condition to broadcast `FINISH` and broadcasting the `FINISH` message is sufficient to ensure liveness.

However, this mechanism demands storing the outputs of all ABBA instances, leading to unbounded memory usage. Thus, the agreement component only retains the outputs of ABBA instances for the last `RetainedEpochCount` epochs (`RetainedEpochCount`×`EpochLength` agreement rounds) and relies on a global checkpoint-based recovery mechanism to maintain liveness. Concretely, this retention limit on ABBA outputs may lead to nodes getting stuck on old ABBA instances (due to the rest of the system having already moved on and cleared the outputs of the old rounds). In this situation, the node eventually receives a valid checkpoint from the orchestrator to recover. Upon receiving the checkpoint, the agreement component advances the Modring window to the first agreement round after the checkpoint, discarding all ABBA instances and their outputs for epochs before the checkpointed epoch minus `RetainedEpochCount`.

4 Optimising Alea-BFT

Contents

Throughout our work, we devised several optimisations to improve the performance of Alea-BFT and the Mir/Trantor framework itself. In this chapter, we present these optimisations, beginning with the optimisations to Alea-BFT and its sub-protocols, followed by how we expedite threshold signature reconstruction and Mir/Trantor-specific optimisations.

4.1 Binary Agreement Unanimity

The binary agreement sub-protocol (ABBA) in Alea-BFT possesses a validity property which prevents a value exclusively proposed by Byzantine nodes from being decided. We exploit this property to expedite ABBA when all nodes input the same value and deliver a result in a single round of all-to-all broadcasts.

Concretely, we augmented the ABBA sub-protocol with an `INPUT` message, which replaces the `INIT` message in the input step of the protocol. This message is identical to the `INIT` message but is only sent by nodes that have not yet input a value into the sub-protocol. If a node receives N identical `INPUT` messages, it can conclude that all correct nodes have provided the same input b to the ABBA. Therefore, b is the result of the sub-protocol (validity property), and the node can immediately deliver b.

However, some nodes may see a different non-unanimous set of ABBA inputs due to Byzantine behaviour. Thus, ABBA can deliver a unanimous input but not immediately terminate: it must continue executing the protocol to ensure other correct nodes eventually deliver. To expedite termination, nodes broadcast the `FINISH` message when delivering through this fast path. To distinguish delivery from termination, we augmented the ABBA implementation with a new `Done` event, which signals when the sub-protocol terminates. This event is used by the agreement component to only mark ABBA instances for garbage collection after they terminate (and not after they deliver).

4.2 Eager Agreement Input

Alea-BFT achieves good performance when compared to other BFT protocols such as HBBFT by moving from a fully leaderless ACS-based design to a simple binary agreement primitive that only decides the delivery of proposals from a single node at a time. However, this may be a bottleneck to scalability as we decide on the delivery of a single proposal at a time. To address this, we explore the possibility of having multiple agreement rounds executing in parallel.

Parallel agreement round execution is possible as long as we can determine what to input to the parallel rounds. We noticed the current agreement round and the next $N - 1$ are all led by different nodes/decide the delivery of different queues (assuming a round-robin policy). Thus, we can map those agreement rounds to the queue slots whose delivery is being decided and determine what the input for those agreement rounds ("1" if the slot was delivered by the broadcast component, "0" otherwise)[1]. Additionally, we must buffer and serialise agreement round outputs to ensure they are delivered in order.

To utilise this mechanism in our implementation, we extend the queue selection policy with a revered version of the aforementioned partial map (from queue slots to agreement rounds) and modify the orchestrator to, after a broadcast delivers, input 1 to the corresponding agreement round if no input was previously provided.

However, multiple ABBA instances executing in parallel comes at the cost of network bandwidth. Thus, we limited the number of eager inputs the agreement component can broadcast to a configurable parameter *MaxAgRoundEagerInput* and buffer agreement inputs, allowing the orchestrator to be oblivious to this limit. Additionally, we opted to inhibit the *MaxAgRoundEagerInput* future agreement rounds from making any progress outside of the unanimity fast path. Concretely, we only allow each of the *MaxAgRoundEagerInput* future rounds to do one of two actions after broadcasting their `INPUT` messages: deliver using the unanimity fast path and broadcast `FINISH` or wait for the previous rounds to deliver before continuing non-unanimous (slow path) ABBA protocol execution.

4.3 Pipeline Tuning

Alea-BFT promises efficiency with a simple design, but a disordered execution of the broadcast and agreement sub-protocols may compromise this efficiency. We identified two main issues that may arise from a disordered execution of the sub-protocols and propose a mitigation for each issue based on delaying protocol steps:

- **Non-Delivering Agreement Rounds.** When the system is under load, we expect all nodes to continuously create and disseminate new batches of transactions. In this situation, if all nodes are correct, zero-deciding agreement rounds are purely wasted work, as they could decide to deliver a batch if only nodes waited for the corresponding broadcast to deliver. Additionally, uniform inputs help ABBA converge faster: if an input is proposed by $\leq F$ nodes, ABBA discards it. Thus, we wish to avoid inputting 0 to ABBA instances to aid ABBA convergence and avoid deciding against delivery.

  Mitigation: **Agreement 0-Input Delay** – delay "0" inputs to agreement rounds when the corresponding broadcasts are estimated to complete soon and when no queues have batches ready to deliver.

- **Poor Quality Broadcasts.** When a client sends the same transaction to multiple nodes, it may be included in different batches by different nodes and thus broadcast several times. While Alea-BFT

[1]This mapping should be possible using other queue selection policies, but must be constructed in a case-by-case basis for each policy.

specifies a deduplication mechanism, it focuses on the agreement component and does not prevent the broadcast component from sending duplicate transactions. However, we can extend the mempool to eliminate duplicate transactions as they are delivered. For this mechanism to work optimally, we wish to create new batches as late as possible, thus maximising the chances that potential duplicate transactions are proposed by other nodes and delivered before being included again in a different batch by a different node.

Mitigation: **Batch Creation Delay** – delay the creation of each new batch such that the estimated time of broadcast completion on all nodes is just before the estimated time when nodes vote on the delivery of that batch.

To ensure the protocol remains responsive to changes in system conditions (e.g., network latency or processing speed), the orchestrator re-evaluates the applied delays whenever it processes (a batch of) new events. In the following sub-sections, we describe the conditions that prompt us to delay agreement input and batch creation.

4.3.1 Agreement 0-Input Delay

To increase agreement efficiency, we avoid inputting 0 to agreement rounds. Concretely, we delay inputting 0 to agreement rounds while no queue has a batch ready to be agreed upon or while the corresponding broadcast is in progress and its duration is within the estimated duration for a broadcast. However, this presents three problems.

Firstly, we cannot delay agreement input indefinitely because detecting nodes behind the rest of the system relies on slow nodes continuously trying to advance to the next agreement round. Specifically, up-to-date nodes need to observe slow nodes attempting to provide input to old agreement rounds to mark them as slow and recover them with the latest checkpoint.

Secondly, Byzantine proposers can consistently slow down agreement input by proposing new batches of transactions just before the corresponding agreement round begins. We leave this issue as future work but outline two potential solutions. One potential solution is a complete overhaul of the delay control logic that integrates agreement 0-input and batch creation delay control and penalises consistently late proposers. Another potential solution is extending the early agreement input optimisation to allow future agreement rounds to progress when the current one is delayed. This modification to early agreement input is simpler than a complete redesign of delay control but may not be sufficient to limit Byzantine node influence, particularly with a high number of nodes.

Thirdly, estimating broadcast durations is not trivial. Leader and follower nodes have different perspectives on broadcast duration, and there is a risk of Byzantine nodes arbitrarily inflating estimates, which allows them to arbitrarily delay agreement rounds by exploiting the previous issue in conjunction with this issue. To account for different perspectives, nodes maintain separate estimates for locally-initiated and externally-initiated broadcasts.

To estimate locally-initiated broadcast duration, nodes track the length of these durations (from VCB input to delivery) and compute the 95th percentile across the last *EstimateWindowSize* measured durations. We select the 95th percentile because we believe 0-inputs to agreement rounds to be highly undesirable and thus worthy of a conservative estimate of broadcast duration.

To estimate externally-initiated broadcast duration, nodes track these durations (from the reception of the first VCB message – SEND – to delivery), and, for each proposer/VCB leader, compute the 95th percentile across the last *EstimateWindowSize* measured durations. To avoid Byzantine node influence, we take the $F + 1$-th highest externally-initiated broadcast duration estimate across the estimates for all external proposers as the (uncorrected) externally-initiated broadcast estimate – d_{bc*}. Additionally, we correct this estimate by adding what we call the *bc-done margin* (broadcast-done margin) estimate. This margin corrects broadcast duration estimates using the timing differences in inputs to each agreement round, which we expect to correlate with differences in broadcast delivery times across nodes. Concretely, during each agreement round, each node tracks when every node provides a positive (1) input to the agreement round. When delivering the agreement round, each node registers, if possible, $t_{\textit{local input}}$ (local input time), $t_{\textit{quorum 1-input}}$ (moment when $N - F$ nodes provide a positive input to the agreement round) and $t_{\textit{total 1-input}}$ (moment when all nodes provide a positive input to the agreement round). Since Byzantine nodes can arbitrarily inflate $t_{\textit{total 1-input}}$ by delaying their input to the agreement round, we limit the influence of this parameter in the estimate to *MaxExtSlowdownFactor* times the time taken by a quorum of nodes to provide 1-inputs. Additionally, the sample computation penalises situations where there are not enough 1-inputs to measure $t_{\textit{quorum 1-input}}$ or $t_{\textit{total 1-input}}$, bounded by the latest estimate for d_{bc*}. We present the complete algorithm that computes each *bc-done margin* sample in Listing 4.1.

4.3.2 Batch Creation Delay

To avoid including duplicate transactions in new batches, we delay batch creation such that the new batch is disseminated to the other nodes just in time for the agreement round that decides its delivery. This delay allows the duplicate transactions to be filtered out of the system when they are delivered (after being proposed in other nodes). Concretely, we delay batch creation while the estimated duration of batch creation and dissemination to all nodes ($d_{\textit{local bc all-done}}$) is less than the estimated time until the next agreement round where the batch's delivery is decided ($d_{\textit{until batch ag}}$).

However, delaying batch creation may hinder performance if batch broadcasts finish after the beginning of their corresponding agreement round. Thus, we are conservative with our estimates by assuming agreements are as fast as possible, taking all fast paths and experiencing 5th percentile latencies, and broadcasts as slow as possible, experiencing 95th percentile latencies. Additionally, we only delay batch

```
inputs:
    MaxExtSlowdownFactor
    $t_{\text{local input}}$, $t_{\text{quorum 1-input}}$, $t_{\text{total 1-input}}$
    $d_{bc*}$        ▷ latest uncorrected estimate for externally-initiated broadcast duration
    last margin est for leader    ▷ 95th percentile of last EstimateWindowSize margins for the agreement round leader
    agreement round output

posQuorumWait ← $t_{\text{quorum 1-input}} - t_{\text{local input}}$        ▷ ⊥ if quorum 1-input was not observed
posTotalDelta ← $t_{\text{total 1-input}} - t_{\text{quorum 1-input}}$        ▷ ⊥ if total 1-input was not observed

if posQuorumWait = ⊥ then
    output $min\{2 \times \text{last margin est for leader}, d_{bc*}\}$ ▷ No reference for margin, so take previous margin (for that queue owner) and double it
else if posTotalDelta = ⊥ then
    output posQuorumWait × $(1 + \text{MaxExtSlowdownFactor})$    ▷ No reference from slow nodes for margin, so assume their maximum slowness
else
    output posQuorumWait + $min\{\text{posQuorumWait} \times \text{MaxExtSlowdownFactor}, \text{posTotalDelta}\}$
```

Listing 4.1: *bc-done margin* Sample Calculation

creation when the computed delay is over the estimated minimum network latency (estimation process described in Section 4.3.2.A) to avoid the overhead of the Mir runtime itself from inflating delays. Furthermore, we also restrict batch creation delays to when at least $F + 1$ queues have batches pending delivery, speeding up convergence during system fluctuations, which may temporarily inflate $d_{\text{until batch ag}}$ over $d_{\text{local bc all-done}}$, unduly delaying batch creation in some nodes. These fluctuations always happen when the system starts: in each node, $d_{\text{local bc all-done}} = 0$ until the node proposes a batch but $d_{\text{until batch ag}} > 0$ after a single agreement round.

To estimate $d_{\text{local bc all-done}}$, we take the 95th percentile of the last *EstimateWindowSize* locally-initiated broadcast durations, measured from input to delivery on all nodes. However, these durations must exclude Byzantine node influence, which could arbitrarily delay acknowledging the broadcast to inflate the estimate. Thus, we track the time spent between VCB input and `QuorumDone` event ($d_{\text{vcb quorum}}$) and between `QuorumDone` and `AllDone` events ($d_{\text{vcb done delta}}$) for all locally-initiated broadcasts. Similarly to the *bc-done margin* from the previous section, we limit the influence of slow, possibly Byzantine, nodes on the estimated broadcast duration by measuring each broadcast's duration as $min\{d_{\text{vcb quorum}} \times \text{MaxExtSlowdownFactor}, d_{\text{vcb quorum}} + d_{\text{vcb done delta}}\}$, where *MaxExtSlowdownFactor* is a tunable parameter controlling how slower Byzantine nodes can be in proportion to the remaining nodes.

We estimate $d_{\text{until batch ag}}$ as $N_{\text{rounds}} \times d_{\text{ag round}}$, where N_{rounds} is the number of agreement rounds to take place until the agreement round corresponding to the batch to be broadcast begins[2] and $d_{\text{ag round}}$ is the estimated duration of an agreement round. Since we assume agreement rounds are as fast as possible, we expect all rounds to take advantage of the ABBA unanimity optimisation (described in Section 4.1)

[2]This calculation assumes a round-robin queue selection policy but is generalizable for all policies if they provide a partial mapping from agreement rounds to the slots whose delivery is being decided.

and thus require a single communication step to deliver. Therefore, we set $d_{ag\ round}$ to the minimum network latency, which we estimate as described in Section 4.3.2.A. Additionally, some agreement rounds deliver instantaneously due to the combination of the ABBA unanimity and the agreement eager input optimisations (described in Section 4.2 and Section 4.1 respectively). To account for instant rounds, we subtract the agreement rounds whose corresponding batch is already locally present from N_{rounds}, on the expectation that all other nodes also have them present and provide inputs to the corresponding agreement rounds.

4.3.2.A Estimating Minimum Network Latency

We devised two methods for estimating minimum network latency based on VCB and ABBA step timings. While we expect the ABBA-based method to be more precise, it relies on ABBA instances to use the slow path, which we do not expect to happen unless some nodes are faulty. Thus, we created the less precise VCB method to provide preliminary estimates of network latency when the ABBA method is unfeasible. To combine the estimates obtained from the two methods, we select the minimum value of the two. In turn, both methods compute their estimate similarly to the estimates described in previous sections: by selecting the 5th percentile of the last *EstimateWindowSize* latency samples. We use the 5th percentile to select a small latency of the recent past while excluding outliers.

To estimate minimum network latency from ABBA instances, we measure, in each ABBA round, the time passed between broadcasting the `AUX` message to when the coin toss starts (right after receiving enough `CONF` messages), which we expect to correspond to roughly 2 communication steps (`AUX` and `CONF`). Therefore, we divide the measured duration by 2 to obtain a network latency sample.

To estimate minimum network latency from VCB instances, we consider locally initiated broadcasts and measure the time difference between the local node's `ECHO` message and `ECHO` messages received from other nodes. This difference corresponds to two communication steps and one threshold signature share creation. Our experiments showed this signing operation to be fast, specifically under $3ms$ in 95% of measured signings as detailed in Section 5.2.1. Therefore, we divide the measured difference by 2 to obtain a network latency sample and assume the signing operation (half-)duration is negligible.

4.4 Fast Path for Threshold Signature Reconstruction

Reconstructing a t-threshold signature requires collecting t valid shares. Validating each signature share is computationally expensive and hinders system scalability, so some systems opt to aggregate the shares and validate the aggregate signature. However, we believe we can achieve a better improvement by validating only the full threshold signature instead of all signature shares. If the full threshold signature is valid, we can safely output it by definition. Conversely, if the full signature happens to be invalid, we fall back to validating the shares individually.

This approach introduces a potential problem: Byzantine nodes can slow down the system by providing invalid shares. However, when we encounter an invalid share, we can safely conclude that the sender is Byzantine and ignore its shares in the future. This exclusion mechanism was not implemented, but it is a straightforward modification. Furthermore, aggregate share verification could be leveraged to improve the slow path.

4.5 Timer Coalescing

The pipeline optimisations introduced in Section 4.3 rely heavily on timers to delay protocol steps. However, our preliminary evaluation showed that nodes spent significant computational resources setting up, firing and processing timers. To address this, we refactored the orchestrator logic to be timer-invariant: rather than scheduling an action to execute when a timer expires, we always execute the timer-associated logic after processing a batch of events and allow it to request the orchestrator to run again after some minimum amount of time. After processing all events, the orchestrator sets up a single timer for the lowest requested wait (or no timer if no requests were made). This timer sends a heartbeat event to the orchestrator, which executes all timer-associated logic again, repeating this process for as long as required.

4.6 Mir/Trantor Optimisations

We conclude this chapter with the set of optimisations we developed during the course of this work that can be applied to the Mir and Trantor frameworks themselves.

4.6.1 Prioritisation of State Transfer Messages

The Mir/Trantor network stack buffers outgoing messages in one First-In First-Out (FIFO) queue per destination. However, these queues may delay or even prevent the delivery of state transfer messages, which are crucial for fast-forwarding nodes stuck in an old system state (epoch). To address this, we augmented the network stack with a `ForceSend` operation that bypasses the queue and is processed as early as possible. Concretely, we introduce a special 1-element queue for `ForceSend` messages that is always processed before the main FIFO queue. This operation should only be used for the latest state transfer message, as repeated calls to `ForceSend` replace the message in the special queue.

`ForceSend` not only speeds up the recovery process under strained conditions but also fixes a potential liveness problem in Mir.

4.6.2 Avoiding Event Conversions

For historical reasons, Mir/Trantor defines all events as protobuf messages, but the more expressive Mir DSL module infrastructure (Section 3.3.4) uses custom Go structures generated from the protobuf message definitions. This mismatch results in unnecessary conversions between protobuf and custom structures every time Mir DSL modules are used: the DSL module must convert protobuf-formatted events to the custom structures to process them and then must convert the output events to the protobuf format. To address this, we changed all event processing code to use the custom structures and only leverage protobuf when serialising messages to the network.

4.6.3 Slices as Event Lists

Mir/Trantor uses linked lists to store events, which are constant-time for appending new elements and can be constant-time for in-place concatenation. At first glance, this seems like a good choice for event lists, as events are constantly appended to module input event lists (from module output event lists) and are often concatenated when combining events from different sub-modules. However, linked lists have poor cache locality and there is a non-negligible cost associated with the indirection of linked lists. In practice, Mir event lists are too small to benefit from the constant-time appends. Additionally, Golang's list implementation does not reuse allocations, forcing new list nodes to be allocated for every append (even if moving an element from another list) and forcing the full list to be copied for concatenation. Thus, we switched the event list representation to a slice of events, which is more cache-friendly and avoids the indirection cost of linked lists.

4.6.4 Module Goroutine Pools

Mir/Trantor provides helpers for creating simple stateless modules – modules where the reaction to an event depends purely on the event itself. These helpers can create passive modules that process events sequentially, by simple iteration and transformation of input events into their outputs, or concurrently, by launching a Goroutine for each event and collecting the results of processing of the batch of events before returning. The concurrent variant is particularly useful for computationally intensive operations (such as signature verification), as it can leverage multiple CPU cores.

However, there is a performance overhead in launching a Goroutine (and a corresponding output channel) for every single operation. To address this, we built a new abstraction based on an active module that processes events concurrently using a fixed-size pool of Goroutines that all output to the active module's output channel. To avoid breaking simulation-based tests, which need special adaptations to work with active modules, we opted to initially create a passive module that processes events sequentially but allow it to be upgraded to the concurrent variant with a single function call. With this approach, tests can benefit from the simplicity (and determinism) of passive sequential modules while production code can leverage multi-core architectures. Additionally, we introduced a new operation to the module set type that upgrades all passive modules using their abstraction to their concurrent variants. This operation is called at the start of the Mir benchmarking code to ensure performance is evaluated with the concurrent modules.

4.6.5 Avoiding Allocations in Event Loop

During Alea-BFT's performance evaluation, we noticed that the Mir/Trantor event loop was one of the biggest Golang garbage-collection stressors, performing many allocations at every iteration. Upon further inspection, we noticed that the event loop was dynamically constructing a Golang select statement at every iteration. This select statement considers all (root) modules that have pending input events, and selects a module that can accept input events to process. To construct it, Mir creates a slice of `SelectCase` objects, each corresponding to the input channel of a Mir module with pending input events. Then, Mir uses Golang's reflection Application Program Interface (API) to perform the select operation, which returns the index of the selected case. Mir then uses this index to call the corresponding closure in a parallelly constructed slice of select reactions.

While it is hard to avoid constructing the select statement dynamically (knowing the set of root modules at compile-time is non-trivial), the set of select cases (and reactions) is always a subset of the set of root modules, which is fixed at startup. Thus, we refactored the event loop to reuse the allocated objects (the slices of select cases and reactions) and avoid duplicate work. Concretely, we start the event loop by constructing a slice with all possible cases and reactions (one per module). In each iteration, we reorder the slices of select cases and reactions in unison, moving all the cases and respective reactions for nodes without pending input to the ends of their respective slices. This reordering brings no additional cost – it replaces the existing linear pass through all root modules to select the ones with pending input events. Importantly, the reordered slice has the desired select cases for the current iteration contiguous at the beginning of the slice. Thus, when calling Golang's reflection API to perform the select operation, we can construct a slice of the desired select cases at no cost, by sub-slicing the slice with all possible cases.

5 Evaluation

Contents

In this chapter, we present a comprehensive evaluation of our Alea-BFT implementation. Section 5.1 describes the setup used in our experiments. Section 5.2 provides a detailed performance breakdown of our implementation, focusing on the latencies of the main sub-protocols and the delays introduced by the optimisations. Section 5.3 compares Alea-BFT with a performant partially-synchronous BFT protocol – ISS-PBFT.

This evaluation aims to answer the following questions:

1. Does the observed performance of our implementation of Alea-BFT match analytical expectations?
2. What are the performance bottlenecks of our current implementation?
3. Is the performance of Alea-BFT competitive with partially-synchronous BFT protocols (specifically ISS-PBFT)?

5.1 Experimental Setup

Over the course of our work we used two different setups: one optimised for fast iteration and prototyping and another optimised for stability, which was used to produce the final evaluation results presented in the remainder of this chapter.

For prototyping and testing, we used the lab machines from Instituto Superior Técnico (specifically the ”RNL Cluster”[1]). To facilitate fast iteration, we built a set of scripts [47] that parallelise test execution across the available machines, with care to isolate each experiment (with potentially different software versions) from the others. Furthermore, this set of scripts recorded all the necessary information to reproduce each experiment, such as the software versions, node configuration and the scripts themselves. Since we tested several experimental configurations with different software versions at the same time, this record was crucial for tracing results back to their respective configurations. While this setup allowed fast iteration during early development and sported a large number of physical machines, it was not a sufficiently stable environment for gaining good confidence in the evaluation results and enabling reproducibility (the cluster machines can be used at any time by anyone), which prompted us to switch to a more controlled environment.

For the final evaluation, we moved to a set of dedicated machines at the INESC-ID cluster, housed in a climate-controlled server room. This setup is composed of 7 physical machines with $128GB$ of RAM connected by a $1Gbps$ network with $0.1ms$ inter-machine latency. Regarding CPU, 3 of the machines possess a 16-core AMD EPYC 7282 processor, while the remaining 4 have a 12-core AMD EPYC 7272 processor. To normalise the number of CPU cores across all machines and allow scalability experiments with high node counts, we run all (logical) nodes on Docker containers limited to 4 CPU cores each

[1] https://rnl.tecnico.ulisboa.pt/servicos/cluster/

and distribute them across the 7 physical machines. To evaluate performance under varying network conditions, we used `tc`'s `netem` to emulate different inter-node latencies (Local Area Nework (LAN)/$0ms$, $5ms$, $25ms$, $75ms$). Additionally, scalability experiments use `tc`'s `tbf` to limit the outbound bandwidth available to each container to $50Mbps$ to avoid different maximum bandwidth between co-located nodes and nodes in different machines. The bandwidth limitation severely constrains the maximum throughput achievable in experiments, so we only enable it for scalability experiments. Each experiment ran for 5 minutes and was repeated 5 times. The full list of configuration parameters for each protocol under test can be found in Appendix B.

In the remainder of this section, we describe how we submit transactions to the system during experiments, and how we measure and present the various properties of the system.

5.1.1 Transaction Submission Method

There are multiple methods to apply load to an SMR system. Clients can either be completely external to the system, submitting transactions through a network interface, or co-located with the nodes, submitting transactions using some form of inter-process or inter-thread communication. Besides co-location, each client can either wait for each transaction to be delivered before submitting the next one – known as a closed loop client – or submit transactions continuously at some pre-determined rate of submission – known as an open loop client.

Initially, Mir/Trantor included support for receiving transactions from external clients and a pre-built generic open-loop client, which we planned to use because it would be a simple way to provide an identical workload to the existing ISS-PBFT implementation. However, during our work, Mir/Trantor switched their primary benchmarking infrastructure to closed-loop clients co-located with nodes (in the same process) for the experiments.

After discussing the matter with the authors, we decided to also switch to closed-loop clients, as they are more appropriate for latency measurements (they impose minimal load on the system) and facilitate peak throughput measurements. Concretely, closed-loop clients include back-pressure by design, which allows measuring peak throughput with a small number of test configurations, whereas using open-loop clients requires increasing system load very slowly to observe peak throughput without overloading the system. While closed-loop clients also eventually reduce system throughput (when used in high enough numbers), this reduction is less pronounced than in open-loop clients, which submit transactions at a fixed rate regardless of the system's ability to process them.

5.1.2 Measuring Peak Throughput

Peak throughput is the maximum throughput observed in the system. Thus, we choose a number of closed-loop clients that aims to keep a small number of full batches of transactions ready to process in each node – namely a number of clients that is a small (1 to 16) multiple of the batch size (B). Concretely, by instantiating, e.g., $4 \times B$ closed-loop clients per node, each client is continuously attempting to order one transaction in the system. Therefore, at any given time, we expect B clients to have their B transactions being processed and the remaining $3 \times B$ clients to have their $3 \times B$ transactions waiting in the system's mempool but ready to be processed as soon as a new batch is requested. However, high client counts can overload the system since each client requires a dedicated Goroutine. Therefore, we needed to conduct some preliminary runs to experimentally determine the optimal number of clients per node for measuring peak throughput for each batch size (B). For experiments with $B = 1$, we observed the optimal number of clients to be $8 \times B = 32$ per node for both protocols: both require at least $2 \times B$ clients to avoid stalls and performed well in all configurations that satisfy this, but $8 \times B$ is the configuration where both protocols performed at their best (by a small margin). For experiments with $B = 1024$, we observed the optimal number of clients to also be $8 \times B = 8192$. For experiments with variable batch size, we collected data for configurations with $1 \times B$, $2 \times B$, $4 \times B$ and $8 \times B$ closed-loop clients per node. For each batch size and protocol, we selected the client count configuration that yielded the highest throughput (averaged across all nodes/repetitions) and discarded the rest. Despite the potential for stalls, it was crucial to test configurations with $1 \times B$ clients because the overhead of Goroutines (one per client) exceeded the overhead of protocol stalls in configurations with significant batch sizes (e.g., $B = 16384$). These configurations with large batch sizes would likely benefit from an alternate client implementation that uses fewer Goroutines to submit the same number of transactions but keeping the closed-loop nature of the clients.

For each combination of experimental parameters, we report the average of all throughput measurements across all repetitions and nodes as a single data point and the respective standard deviation as error bars.

5.1.3 Measuring Base Latency

Base latency represents the time for a transaction to be processed by the system in optimal conditions. To measure base latency, we set the batch size to 1 and create two closed-loop clients on each node. While base latency measurements demand an unloaded system, using a single closed-loop client for latency measurements leads to stalls in both protocols. Thus, we increased the number of clients to two per node, which ensures that each node always has at least one transaction to process, preventing stalls without imposing a big load on the system, which would add noise to our measurements.

However, using two clients inflates latency measurements – the latency of one transaction includes part of the latency of the previous transaction. To exclude the effects of the two-client setup on the results, we measure transaction latency from the time the transaction is removed from the mempool (instead of client submission) to the time it is delivered to the client. This method measures the optimal latency of the system, emulating a situation where transactions are submitted at the exact moment the system is ready to start processing them.

For each combination of experimental parameters, we report the average latency measurement across all repetitions and nodes as a single data point and the respective standard deviation as error bars.

5.1.4 Measuring Sub-Protocol Latency

To measure sub-protocol latency, we introduced a custom event interceptor to each node, which tracks the start and end of various protocol steps. The specific events that determine when a protocol step starts/ends vary widely, so they are presented together with their relevant results. This interceptor was tailored to our Alea-BFT implementation and imposes significant overhead due to the high number of events it tracks (several millions for each 5min experiment repetition in LAN settings). Thus, this interceptor was only active for collecting data on sub-protocol latencies (presented in Section 5.2) and not for macro-level base latency or throughput measurements.

Experiments using the interceptor all ran in a fault-free environment with a batch size $B = 1$, across $N = 4$ nodes with 2 closed-loop clients each, configured to tolerate up to $F = 1$ Byzantine faults. We repeated these experiments under various network conditions (LAN, $5ms$, $25ms$, $75ms$). We opted to exclude LAN results due to the high number of outliers in our experiments, which was likely caused by CPU contention (LAN experiments achieve higher throughput, meaning they require more computational resources).

Sub-protocol latencies are presented using box plots, where the whiskers represent the 2.5th and 97.5th percentile of the dataset, meaning we consider the outermost 5% of our data points as outliers and discard them.

5.2 Alea-BFT Execution Breakdown

Alea-BFT relies on sub-protocols for disseminating transactions, agreeing on their delivery and recovering from faulty/slow proposers. To understand the behaviour of this protocol, we break down processing into these various stages, including the aforementioned sub-protocols and the pipeline optimisation delays described in Section 4.3. We used the trace data collected by the custom event interceptor and experimental setup described in Section 5.1 and aggregated it into five different stages:

- Batch Creation Delay – Delay stalling the creation of a new batch (pipeline tuning optimisation).

- Broadcast – Duration of broadcast stage: measured from the moment a new batch is requested from the mempool (or the first received VCB message, in the case of VCB followers) to VCB delivery.
- Agreement 0-Input Delay – Delay stalling 0 inputs to agreement rounds (pipeline tuning optimisation).
- Agreement – Duration of agreement stage: measured from the end of the previous agreement round (excluding 0-input delay) to the current round's delivery.
- Batch Fetching – Duration of batch fetching stage: measured from agreement round delivery to application delivery. This stage is Trantor-specific and includes Alea-BFT's recovery protocol and delays related to serialising application deliveries with state snapshot requests (used for checkpointing).

Despite having collected data for multiple inter-node latencies, we initially focus on the WAN scenario ($75ms$ of inter-node latency) to minimise visual noise (fewer transactions are processed per unit of time) and computational overhead from cryptography. We select the median-latency transaction from this data set and present it as an execution trace (Figure 5.1) broken down into stages from the perspective of node 0. In addition to the median-latency transaction, the trace includes the 8 transactions that precede it and the 8 transactions processed after it. Each tick in the Y axis corresponds to a transaction (identified by its proposer/queue owner and sequence number/queue slot) from the perspective of a node. Time is represented in the X axis with the conventional direction (left-to-right) and is normalised: $t = 0s$ corresponds to the moment the median-latency transaction (`2/638`, highlighted with darker colours) leaves the mempool in its proposer (node 2). The median-latency transaction (`2/638`) is highlighted by darker shades of all colours.

We will now describe the processing of the median-latency transaction from the perspective of node 0. This transaction is proposed by node 2 and is first seen by node 0 $t \approx 82ms$ (≈ 1.1 communication steps), which corresponds to some pre-processing of the new batch and the time to receive VCB's `SEND` message from node 0. The broadcast stage continues and delivers in node 0 at $t \approx 263ms$ (≈ 3.5 communication steps after $t = 0$). Thus, our broadcast implementation incurred approximately $37.5ms$ of overhead. After broadcast delivery, the system waits for approximately $75ms$ (≈ 1 communication step) before being able to start the agreement round for this transaction. This delay is not the agreement 0-input delay but rather is imposed by the previous agreement round, which takes this time to complete. Finally, the agreement round that decides the delivery of this transaction starts and lasts under $15ms$ (under 0.2 communication steps). The transaction is almost instantly delivered to the application afterwards.

Regarding other transactions, they display similar results, apart from the duration of agreement rounds and related (non-optimisation-related) delays, which vary periodically.

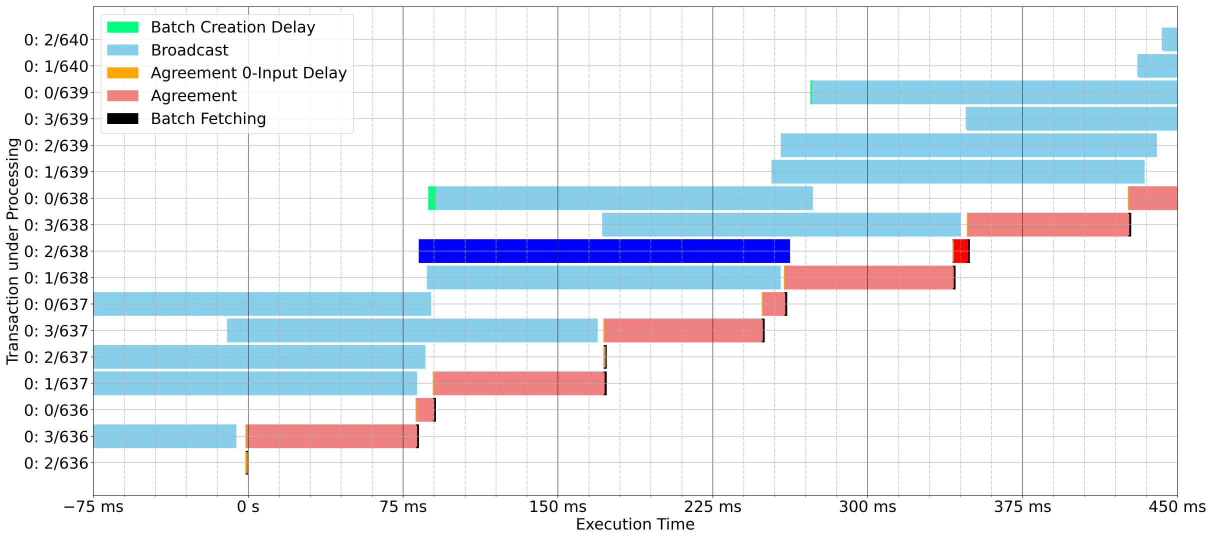


Figure 5.1: Alea-BFT Execution Trace (node 0)

To ensure the observed behaviour is not exclusive to node 0, we also produced an execution trace that shows all node perspectives (Figure 5.2), albeit for a shorter time window. Each set of bars on the Y axis corresponds to a transaction from the perspective of a single node, where all transactions are ordered by time of delivery and node perspectives are ordered by node ID. To aid readability, only the bars for the node 0 perspective are filled. The remaining visual elements follow the same design from the previous trace in Figure 5.1.

This global perspective paints a similar picture to the previous analysis. For most stages, all nodes display similar timings in every transaction. The only exceptions are the batch creation delay stage, which only exists in the proposer node, and the broadcast stage, which is driven by the proposer and thus starts one communication step earlier in this node. The broadcast stage also delivers earlier in the proposer node due to the nature of the VCB protocol.

In the following sub-sections, we further analyze this trace focusing on each of the aforementioned stages.

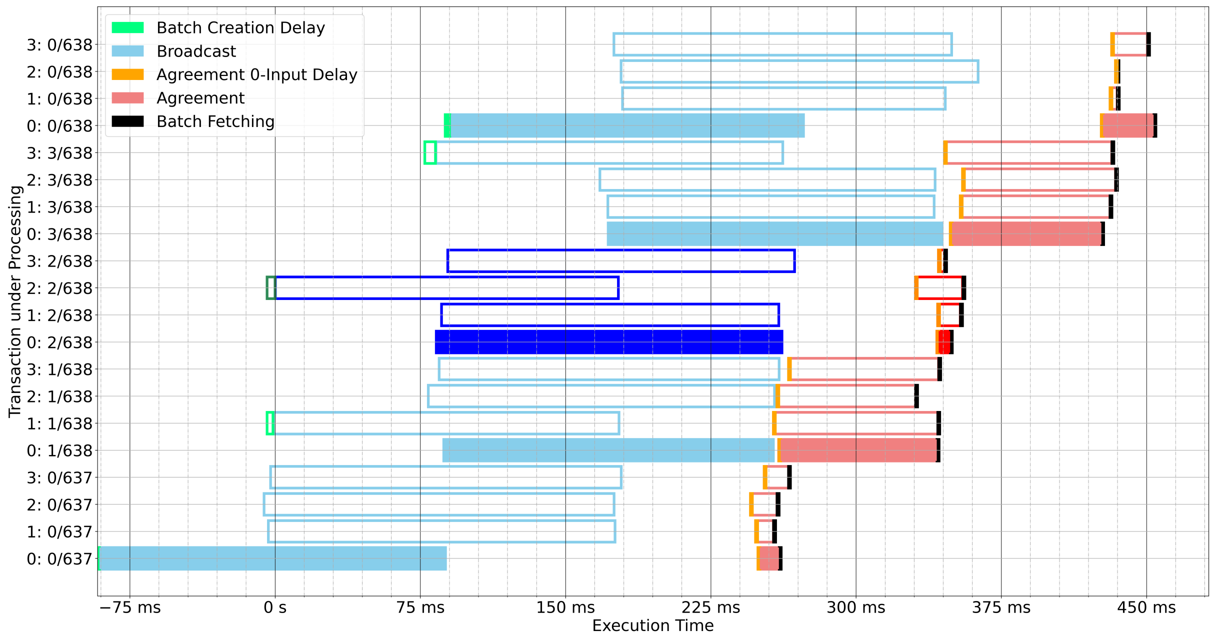


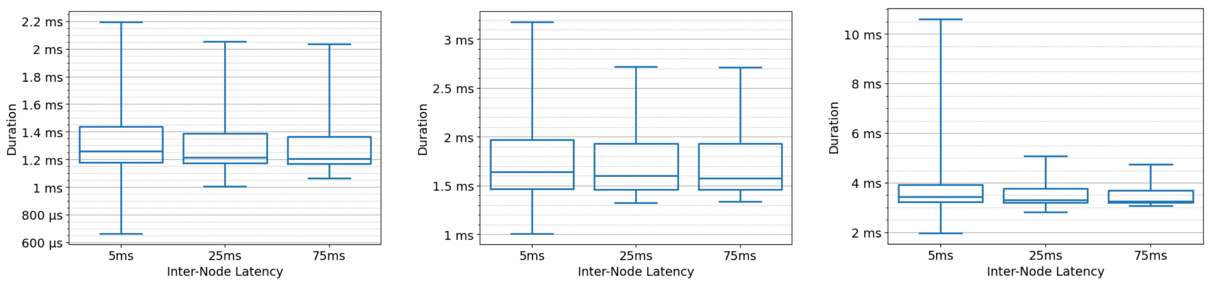
Figure 5.2: Alea-BFT Execution Trace (all nodes)

5.2.1 Broadcast Latency

The broadcast stage encompasses obtaining a new batch of transactions from the mempool and broadcasting it using VCB. During this stage, node perspectives differ. The broadcast leader obtains the new batch of transactions from the mempool, stores it and delivers after just two communication steps – simultaneously broadcasting `FINAL` to other nodes. Broadcast followers only observe the broadcast after the first communication step of the VCB protocol (receiving the `SEND` message from the leader) but only deliver after receiving the `FINAL` message from the leader and verifying its signature. Thus, broadcast followers only deliver one communication step and signature verification after the leader.

By inspection of the execution trace from all node perspectives (Figure 5.2), we conclude that the median-latency transaction had its broadcast stage last approximately $270ms$ (≈ 3.6 communication steps). Without overhead, we would expect VCB to begin in follower nodes at $t = 75ms$ (after 1 communication step), to deliver in the leader at $t = 150ms$ (after 2 communication steps) and to deliver in all follower nodes at $t = 225ms$ (after 3 communication steps). However, the leader node delivers at $t \approx 180ms$ (after ≈ 2.4 communication steps), and the earliest follower node delivers at $t \approx 262.5ms$ (≈ 3.5 communication steps after starting the broadcast).

We will now attempt to explain this overhead, beginning by analysing the use of threshold cryptography, known for its high computational cost. From $t = 0s$ to leader delivery ($t \approx 180ms$), we expect three cryptographic operations to take place: followers use `SignShare` in parallel to construct signature shares, then the leader uses `Recover` and `VerifyFull` to reconstruct the full signature from shares (following the fast protocol from Section 4.4). After the leader delivers and broadcasts `FINAL`, VCB followers receive `FINAL`, verify the included VCB proof and deliver. Figure 5.3 shows the collected performance data for threshold cryptography operations during the trace experiments, which perform identically under all inter-node latency scenarios. Considering the median durations for each operation, we expect cryptography overhead to be $\approx 1.2ms + 1.6ms + 3.25ms + 3.25ms \approx 9.3ms$, leaving $262.5ms - 225ms - 9.3ms = 28.2ms$ of unexplained overhead.

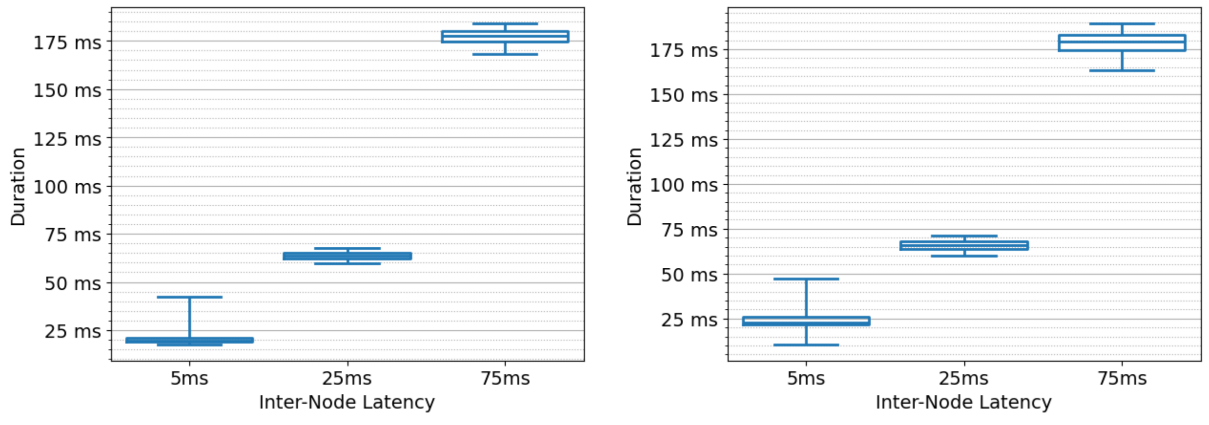


(a) `SignShare` Latency **(b)** `Recover` Latency **(c)** `VerifyFull` Latency

Figure 5.3: Threshold Cryptography Operation Latencies

Despite our expectations, threshold cryptography only accounts for a quarter of the observed overhead. We believe the remaining overhead may be due to inefficiencies in the Mir runtime, which handles all inter-module communication within each node but was not subject to thorough optimisation. Additionally, we believe module hierarchies exacerbate this problem by requiring event lists to be split and merged at every level of the hierarchy.

Finally, we argue that our analysis of broadcast duration and overhead in the median-latency transaction generalises to other transactions due to the consistent broadcast durations measured for leaders and followers, plotted in Figure 5.4.



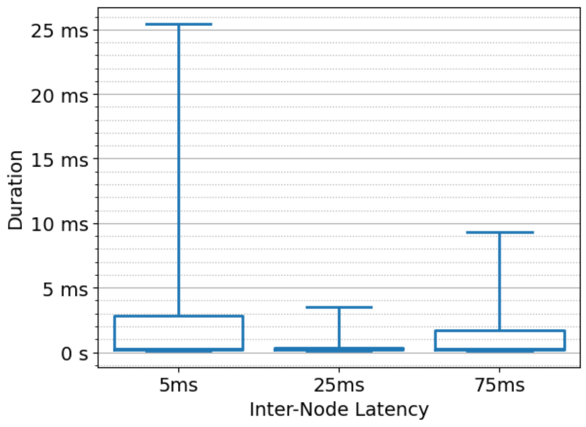
(a) Broadcast Leaders **(b)** Broadcast Followers

Figure 5.4: Alea-BFT Broadcast Stage Duration

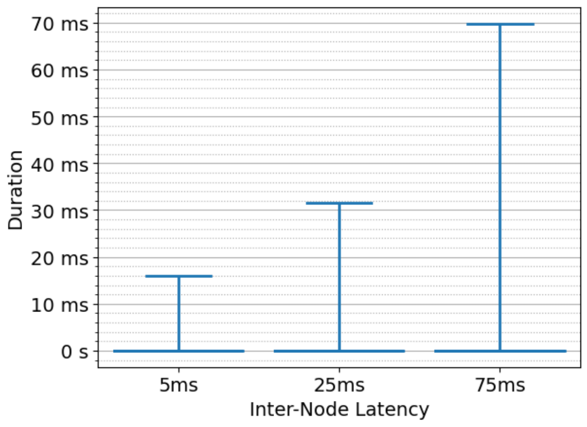
5.2.2 Protocol Step Delays

In the presented traces, we observe negligible delays related to pipeline optimisations. For agreement 0-input delay, this is a positive sign that the pipeline is broadcasting transactions in a timely manner (without delaying input). However, for the batch creation delay, this is problematic since we can also see significant (≈ 1 communication step long) stalls between broadcast delivery and the start of the corresponding agreement round, suggesting it should be increased.

To investigate the matter, we plotted these delays using all of the experiment data (in Figure 5.5) and observed significant delays in the tails of their distributions, particularly in the $5ms$ inter-node latency case. While the 75th percentile of agreement 0-input delay is negligible ($< 1ms$), meaning batch creation was not overly delayed for 75% of transactions, the 97.5th percentile of this metric is just over 3 communication steps long in the $5ms$ inter-node latency configuration, and around 1 communication step long for the $25ms$ and $75ms$ inter-node latency configurations. Thus, 5% of transactions experienced an overinflated batch creation delay. Since the late broadcasts severely compromise protocol efficiency, we believe caution should be exercised in any attempt to increase the batch creation delay.



(a) Batch Creation Delay



(b) Agreement 0-Input Delay

Figure 5.5: Alea-BFT Pipeline Optimisation-Induced Delays

Regarding agreement 0-input delay, it works as intended, avoiding 0-inputs to agreement rounds to encourage the agreement component to perform useful work in every round (decide to deliver rather than not deliver batches/transactions). In the conditions of these experiments (all nodes are correct and have transactions to order at all times), this delay is always beneficial when applied and its existence merely indicates problems in earlier stages.

However, more research is needed to understand the impact of these delays in the presence of faults. Specifically, a faulty proposer may consistently slow down the system by always starting its broadcasts so that the first message arrives just before the corresponding agreement round starts, causing an agreement 0-input delay that spans the duration of a broadcast. The batch creation delay may also hinder performance in the presence of an adversarial network, but we expect its impact to be small when compared with the effects on other protocol stages due to the conservative approach when applying it.

5.2.3 Agreement Latency

Alea-BFT agrees on the delivery of batches of transactions using a binary agreement primitive. Despite this primitive requiring several rounds with 4 communication steps each, the agreement stages shown in the execution traces oscillate between negligible durations and approximately a single communication step. We attribute this small duration to the combination of two optimisations: ABBA unanimity (Section 4.1) and agreement eager input optimisations (Section 4.2). Concretely, the unanimity optimisation brings down the duration of each agreement round to a single communication step, and the eager input optimisation allows the agreement rounds to exchange inputs before their turn, meaning they can start after reaching unanimity and immediately deliver.

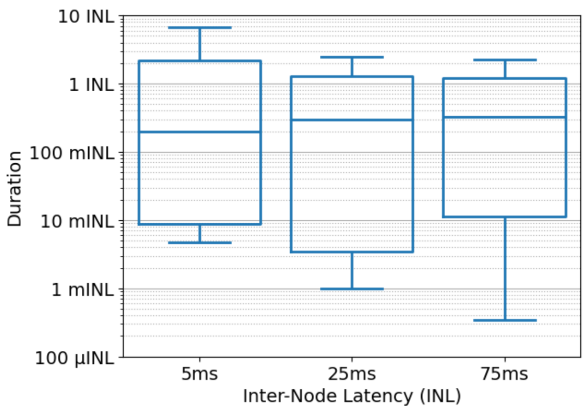
To better understand agreement latency, we divide the remainder of this analysis into three parts. Firstly, we present our full nominal agreement latency results, using not only the traces from Figures 5.1 and 5.2 but the complete collected dataset. Secondly, we describe a potential avenue for pipeline optimisation uncovered during this analysis. Thirdly, we characterise the duration of ABBA rounds to understand if non-unanimous ABBA executions are as expected.

5.2.3.A Nominal Agreement Latency

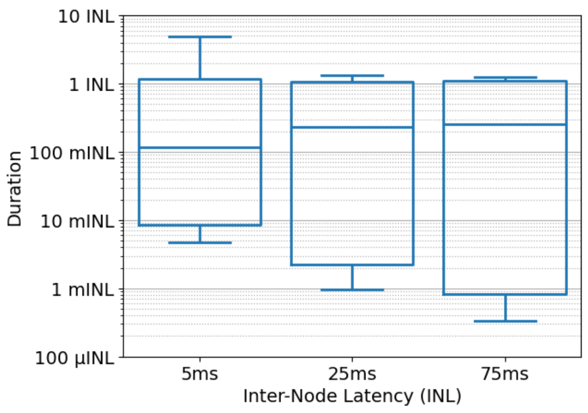
We present nominal agreement latency in Figure 5.6 split between two perspectives: the agreement round leader's (Figure 5.6a) – whose queue head delivery is being decided – and the agreement round followers' (Figure 5.6b) – the remaining nodes. This separation is important because agreement round leaders may provide input to the respective ABBA instance much earlier than followers (the respective batch broadcast delivers around one communication step earlier in the broadcast leader). Additionally, the plots from Figure 5.6 present latencies as multiples of inter-node latency (INL) to aid visualization and in logarithmic scale to show the lower tail of the distribution. From these plots, we extract two key points.

Firstly, agreement round leaders have slightly higher agreement latencies than the remaining nodes. This discrepancy is explained by a similar disparity in agreement 0-input delays: leaders immediately provide input to the next agreement round, whereas followers sometimes stall for a short time.

Secondly, agreement latency is small overall, similar to the transactions present in the trace snippet, which exploit a combination of agreement and ABBA optimisations to achieve latencies under $1INL$ (roughly a single communication step). However, unlike the trace snippet, over half of all agreement rounds have latencies under $1INL$, which indicates better exploitation of agreement optimisations than observed in the snippet. Concretely, for wide-area scenarios ($25ms$ and $75ms$ of inter-node latency), every instance of ABBA terminated in less than $4INL$. Since a single ABBA round requires 4 communication steps, we conclude that all agreement rounds in this configuration were able to exploit the unanimity optimisation. Additionally, between 50% and 75% of agreement rounds delivered in under $1INL$. Thus, several transactions experienced even better exploitation of the eager input optimisation than the ones shown in the trace. The $5ms$ inter-node latency scenario has comparatively high ($\approx 7INL$) tail latencies. We found several agreement rounds that could not take advantage of the unanimity optimisation in this configuration, which explains this disparity.



(a) Agreement Round Leader Latency



(b) Agreement Round Follower Latency

Figure 5.6: Alea-BFT Agreement Latency

5.2.3.B Further Optimising the Pipeline

The remarkable improvement in agreement latency brought upon by the ABBA unanimity optimisation, combined with the eager input optimisation, revealed a previously hidden, but relevant, bottleneck: the serial nature of Alea-BFT's broadcast queues. While inspecting the execution trace snippet, we noticed that broadcasts could no longer keep up with the agreement component: when using the two optimisations, it can potentially deliver batches from all queues at every communication step, but the broadcast component in each node can only place a new batch in the corresponding queue every 3 communication steps. Thus, we want each node to broadcast multiple transactions in parallel, tuning broadcast throughput to match agreement throughput. Preliminary tests of this optimisation in an uncontrolled WAN environment doubled system throughput with a small ($< 10\%$) increase in latency.

5.2.3.C ABBA Slow-Path Latency

While the previously presented results show low agreement latency, this low latency is reliant on the ABBA unanimity optimisation, which can be negated by a single faulty node. Thus, to study agreement latency, we must analyse slow path ABBA latency (i.e., when unanimity is not possible). To characterise slow path performance of our ABBA implementation, we decompose its latency into two variables: the number of ABBA rounds required to deliver and the duration of each ABBA round. Since all agreement rounds in $25ms$ and $75ms$ inter-node latency experiments were able to take advantage of the unanimity optimisation, we focus our analysis on the non-unanimous ABBA trace information from the $5ms$ inter-node latency experiments. From this dataset, we select only ABBA rounds that spanned at least 4 communication steps ($20ms$) to exclude ABBA instances exploiting the unanimity optimisation. This selection corresponds to 47464 ABBA instances, roughly 5% of all observed ABBA instances in this experimental configuration.

Regarding the number of ABBA rounds required for ABBA to deliver, we measured the 75th percentile count to be 1 round and the 97.5th percentile to be 3 rounds. However, this experimental environment does not include faulty nodes and induces a uniform constant load on the system. Thus, in the presence of Byzantine nodes sending carefully crafted messages or asymmetric load across the system causing bigger splits in agreement inputs, the number of rounds per ABBA instance may be higher.

Regarding the latency of each ABBA round (plotted in Figure 5.7), we observed variation between $20ms$ and $44ms$ after excluding the top and bottom 2.5% latencies, with a median of $\approx 26ms$. We expect each ABBA round to take $20ms$, corresponding to 4 communication steps lasting $5ms$ each (the inter-node latency), plus the overhead associated with the shared coin implementation (Section 2.4.3.C), which relies on threshold signatures and was measured to be around $5ms$ (see Figures 5.3b and 5.3c in Section 5.2.1). Thus, the observed median ABBA round latency is in line with analytical expectations.

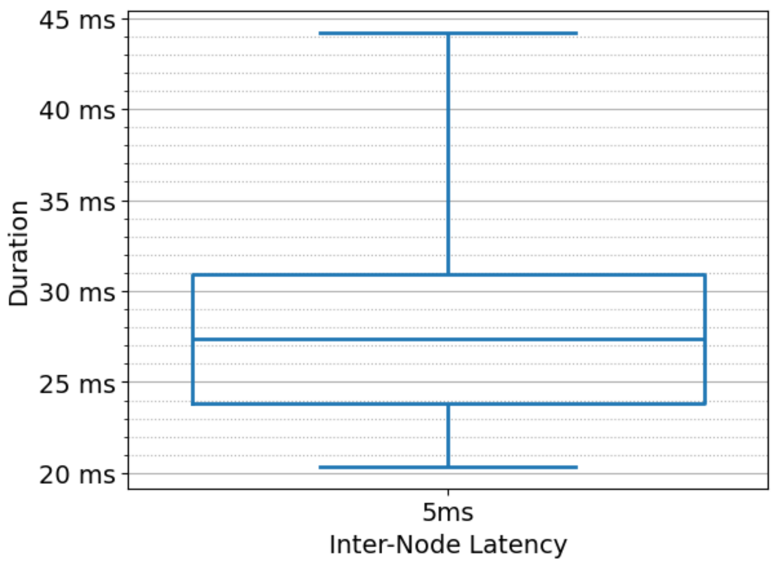


Figure 5.7: ABBA Round Latency

In conclusion, our implementation of ABBA appears to be performing within the analytically expected latency but more research is needed to assess the impact of Byzantine nodes and non-uniform inputs.

5.2.4 Batch Fetching Latency

The batch fetching stage takes place when Trantor requests the ordered transactions from the agreement component to deliver to the application, recovering from faulty/slow proposers if needed and serialises deliveries with snapshotting of the replicated application state for checkpoint creation.

We measured the batch fetching latency and observed a sub-millisecond 97.5th percentile across all experiments. Thus, we conclude that the recovery protocol was never triggered, meaning the broadcast stage and batch creation delay stage operated correctly and on time. Additionally, we confirm that the asynchronous checkpointing protocol does not delay transaction delivery.

5.3 Comparing Alea-BFT with other protocols

Alea-BFT promises simple and performant Byzantine Fault Tolerance on asynchronous networks. However, partially synchronous BFT protocols are known to perform excellently in fault-free runs. Thus, we compare our Alea-BFT implementation to a state-of-the-art partially-synchronous BFT protocol – ISS-PBFT – which was also built on the Mir framework. To ensure our comparison remains as fair as possible, we execute ISS-PBFT with all the developed Mir runtime optimisations (presented in Section 4.6) and configure it with similar parameters to its initial evaluation by Stathakopoulou et al. [6]. We list the complete set of configuration parameters for each protocol in Appendix B.

It is important to note that the evaluated implementation of ISS-PBFT differs from the original publication. Concretely, it is missing the bucket partition mechanism designed to prevent transaction duplication. However, since co-located clients only submit transactions to their respective nodes and the Filecoin subnets follow this co-located client model, this does not pose a problem.

Additionally, base latency measurements in ISS-PBFT are only reported for node 0. We observed the remaining nodes to have inflated latency values, which we attribute to poor interactions of this implementation of ISS-PBFT and our experimental setup. Specifically, we only observed this problem when running experiments inside Docker containers. Since measurements in a Docker-less test environment match node 0 latency in the final experiment environment, we opted to keep the experimental setup the same and only report ISS-PBFT base latencies from node 0, thus avoiding penalising our baseline.

In the remainder of this section, we compare base latency and peak throughput between Alea-BFT and ISS-PBFT varying various parameters. Firstly, we measure performance under various induced inter-node latencies. Secondly, we study the influence of system scale. Finally, we analyse the impact of crash faults on throughput. To aid readability, all plots in this section represent Alea-BFT in blue with circle marks and ISS-PBFT in orange with triangular marks.

5.3.1 Performance Under Varying Network Conditions

To study baseline performance, we performed experiments in a 4-node configuration while varying the induced inter-node latency. In the following sub-sections, we present our results for base latency, choice of optimal batch size and peak throughput.

5.3.1.A Base Latency

Regarding base latency, plotted in Figure 5.8, we observe Alea-BFT to closely follow ISS-PBFT. In low inter-node latency scenarios (LAN and $5ms$), Alea-BFT appears to have nearly constant latency while ISS-PBFT achieves lower latencies overall, particularly in LAN scenarios. We attribute this difference to Alea-BFT's higher computational overhead associated with threshold cryptography and extensive use of Mir module hierarchies. However, as inter-node latency increases, the gap between the two protocols diminishes, and Alea-BFT surpasses ISS-PBFT in the wide-area scenario ($75ms$ of inter-node latency).

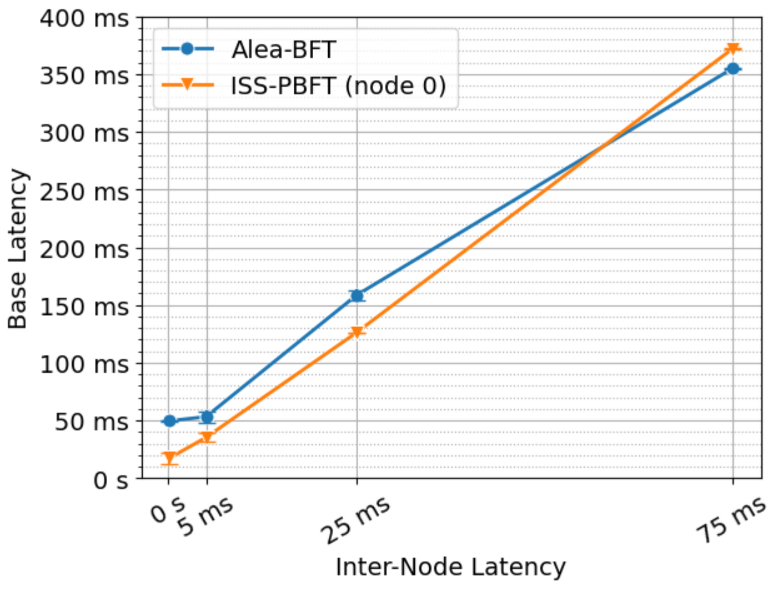


Figure 5.8: Base Latency vs Inter-Node Latency with 4 nodes

5.3.1.B Choosing a Batch Size

SMR protocol throughput is often bottlenecked by the agreement (ordering in ISS) stage of the protocol. To sidestep this inefficiency, protocols often group transactions in batches to order more transactions with fewer agreement stages. In this section, we evaluate the peak throughput of both protocols under study when varying batch size under a 4-node configuration with no induced latency.

Figure 5.9 plots the results of this evaluation, where we observe both protocols' throughput peaking with $B = 8192$. However, using batch sizes beyond $B > 1024$ required modifying the default network

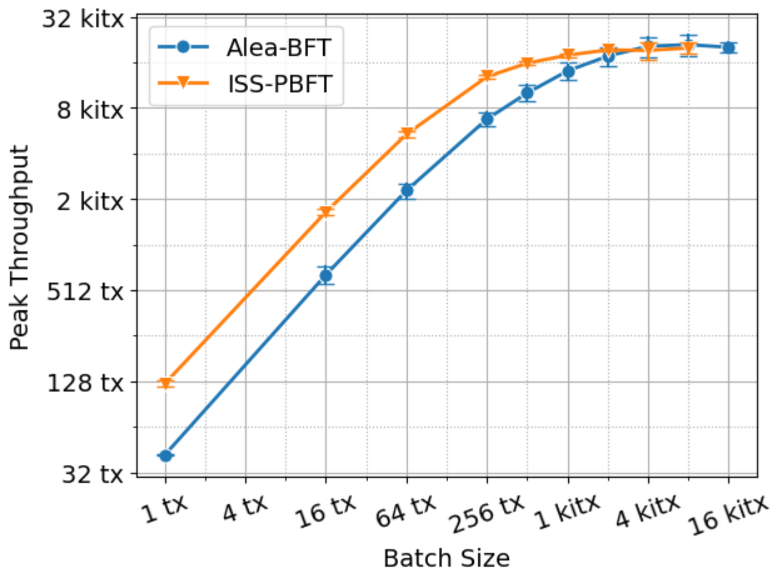


Figure 5.9: Peak Throughput vs Batch Size with 4 nodes in a LAN configuration

outbound message buffer capacity and led to instability in our experiments (the system often stalled indefinitely due to lost messages). Thus, we opt to use $B = 1024$ for throughput measurements rather than the observed optimal setting ($B = 8192$) or the setting used in the original ISS-PBFT evaluation by Stathakopoulou et al. [6] ($B = 2048$).

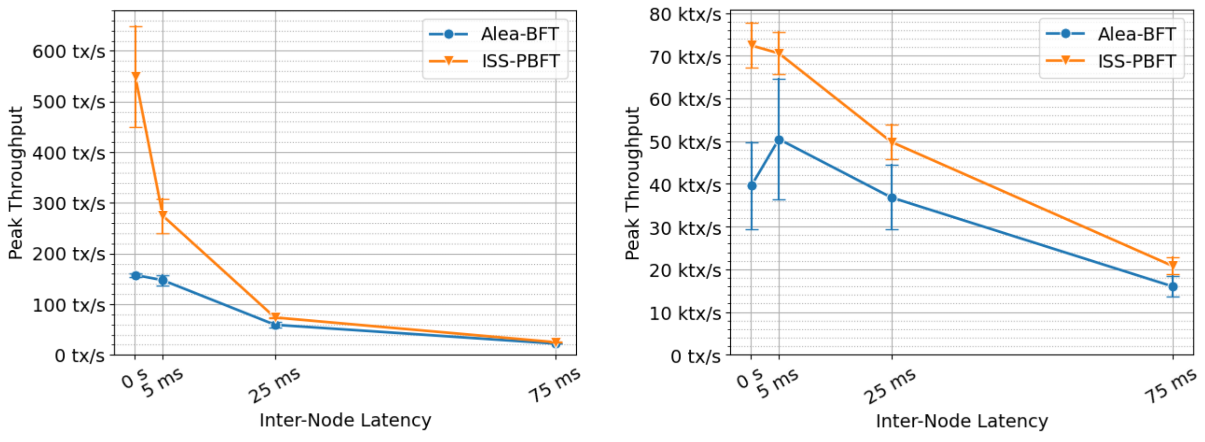
5.3.1.C Peak Throughput

Regarding peak throughput, we studied both protocols under varying induced inter-node latencies using two different batch sizes (B), namely, $B = 1$ (plotted in Figure 5.10a) and $B = 1024$ (plotted in Figure 5.10b).

Despite small batch sizes being inadequate for throughput testing – real-world systems often leverage batching to achieve higher throughput with a comparably smaller sacrifice in latency – we opted to also conduct peak throughput tests with $B = 1$ in this section to assess the throughput capability of agreement (ordering in ISS) component in both protocols. In this experiment, we observed that ISS-PBFT can achieve much greater throughput in low latency scenarios. However, as inter-node latency increases, this difference disappears, and both protocols converge to a very similar peak throughput with inter-node latencies of $25ms$ and $75ms$.

We attribute this gap to the higher degree of parallelism of ISS, which allows each node to broadcast multiple batches of transactions in parallel, unlike Alea-BFT's broadcast component, which proposes new batches sequentially. In the Alea-BFT execution breakdown (Section 5.2), we proposed a modification to Alea-BFT that allows parallel broadcasts (Section 5.2.3.B), which we expect to bring Alea-BFT on par with ISS-PBFT. Alea-BFT's agreement component is also sequential in nature, but the implemented eager input and ABBA unanimity optimizations (Sections 4.1 and 4.2) enable us to parallelise agreement. [2]

With $B = 1024$, the batch size chosen for throughput tests, both protocols' peak throughput is in the same order of magnitude across all inter-node latencies. Similarly to the previous scenario, Alea-BFT's peak throughput converges to ISS-PBFT's and almost matches it in the $75ms$ inter-node latency scenario.



(a) $B = 1$ **(b)** $B = 1024$

Figure 5.10: Peak Throughput vs Inter-Node Latency for various batch sizes B with 4 nodes

5.3.2 Scalability

To evaluate the impact of system scale on performance, we run experiments under various system scales (N) fixing the number of tolerated faults F to its maximum value ($F = \lfloor \frac{N-1}{3} \rfloor$), induced inter-node latency to $25ms$ (to exclude the effects of computation overhead) and batch size to 1024 (the chosen real-world setting in Section 5.3.1.B). In the following sub-sections, we present our base latency and peak throughput results.

5.3.2.A Base Latency

Figure 5.11 plots base latency as a function of system size. Both protocols achieve the same latency for $N < 22$, after which Alea-BFT becomes noticeably slower. We attribute this to ISS-PBFT's multi-leader design, which allows requests to be processed as soon as they reach the PBFT primary node, whereas, in Alea-BFT, we have to wait for the designated node's turn to run its agreement round. Additionally, there is no prioritisation mechanism for threshold signature operations. Thus, the higher number of concurrent

[2] In reality, parallelism can be lost with a single faulty node but we can trivially modify the eager input optimization to allow parallel agreement rounds that do not use the ABBA unanimity fast path.

VCB instances may lead to the broadcast corresponding to some agreement round being prioritised over the broadcast corresponding to an earlier agreement round, inflating latency.

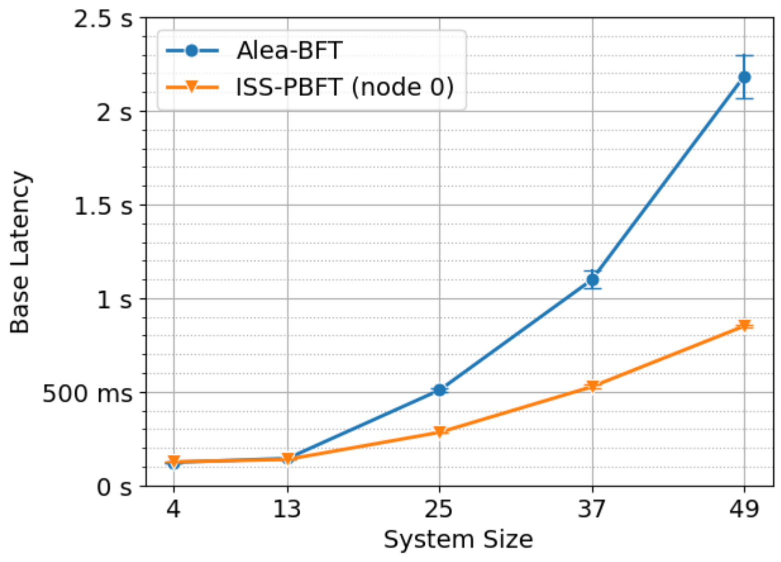


Figure 5.11: Base Latency vs System Size with $25ms$ of induced inter-node latency

5.3.2.B Peak Throughput

Figure 5.12 shows a plot of peak throughput as a function of system size. While Alea-BFT's throughput degrades gracefully, ISS-PBFT's sharply declines with $N = 13$ and approaches 0 for larger system scales. However, we do not believe this is a problem with the design of ISS or ISS-PBFT but rather an implementation issue, for reasons we will now describe.

For $N = 13$, logs reveal the dropping of inbound messages due to size limits on the factory module (Section 3.3.3) being exceeded. In contrast, Alea-BFT, which may have dropped incoming messages, uses the Modring abstraction (Section 3.4.2) to prioritise dropping messages that are expected to be required farther in the future.

For $N \geq 25$, we observed ISS-PBFT to overflow the outbound buffers of the network module, causing the dropping of outgoing messages and, eventually, the system to halt irrecoverably. Alea-BFT likely sidestepped this issue due to the serial nature of its agreement component. Similarly to replacing Factory Module with Modring to avoid the intricacies of selecting buffer sizes, we propose exploring a different network stack design that moves the responsibility of buffering outgoing messages to the sender modules, which avoids the problem of selecting global outbound message buffer sizes.

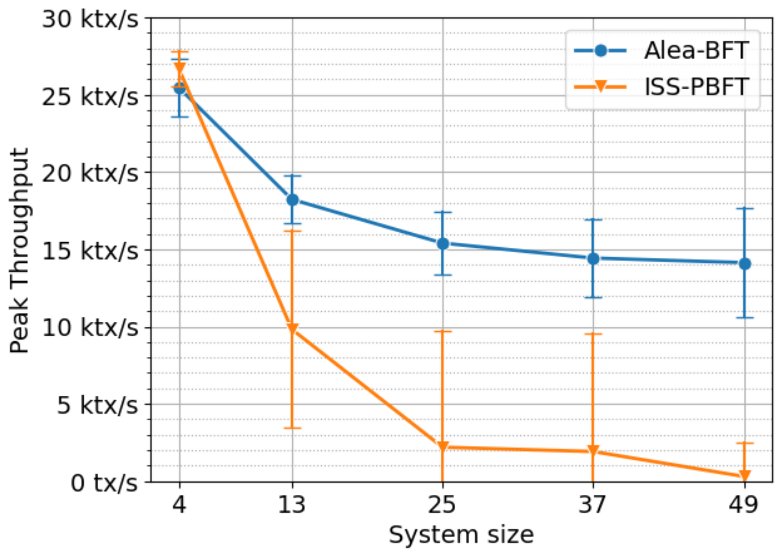


Figure 5.12: Peak Throughput vs System Size with $25ms$ of induced inter-node latency and batch size $B = 1024$

5.3.3 Throughput Under Crash Faults

Finally, we evaluated the throughput of both protocols when facing a crash fault. We performed the relevant experiments in a 4-node configuration connected by a LAN network (no induced latency) with the selected batch size for throughput tests (1024). After $150s$ of runtime, we crash node 0 and never recover it.

Figure 5.13 shows an execution trace of node 1 on one of the five repetitions of the experiment around the moment of the crash for each protocol under study. To aid the readability of pre-crash and post-crash throughput, we also plot a moving average of throughput across all experiment repetitions and nodes as a dotted line – blue for Alea-BFT, orange for ISS-PBFT. 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 node, whereas Alea-BFT can continue uninterrupted (albeit at reduced throughput) thanks to the leaderless design of its agreement component. After this timeout expires, ISS excludes the crashed node from the set of leaders and continues with a relatively small $(\approx 20\%)$ performance hit. However, Alea-BFT is penalised on two fronts – it both loses a node proposing requests (like ISS) and the ABBA unanimity optimization – leading to a reduction in throughput compared to the system with all nodes functional.

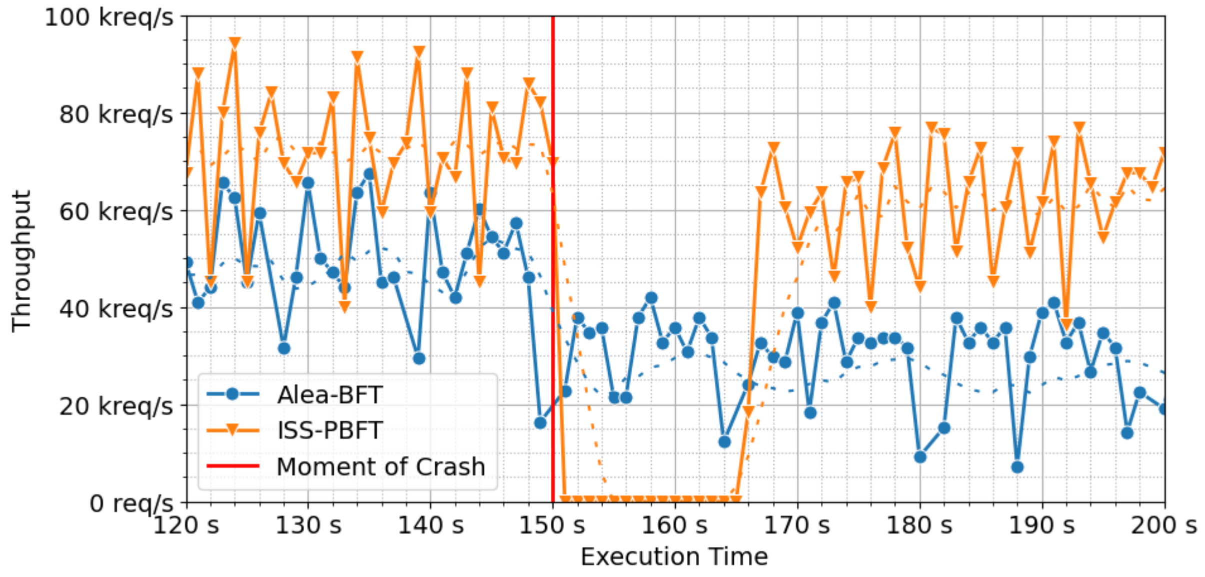


Figure 5.13: Execution Trace in a LAN configuration and batch size $B = 1024$ with a non-recovering crash at $t = 150s$

6 Conclusion

Contents

In this chapter, we summarise the main contributions of this work and outline various avenues for future research.

6.1 Conclusions

This thesis was motivated by the lack of real-world adoption of asynchronous BFT, which is a class of BFT consensus that promises more robustness to performance degradation attacks when compared to their partially-synchronous counterparts.

To bring asynchronous BFT to the limelight, we implemented, optimised and evaluated a practical asynchronous BFT protocol – Alea-BFT – in the context of a real-world system by building it on top of the Mir framework, which the authors intend to become a consensus layer in Filecoin subnets. During this process, we were concerned with bounding resource usage of all protocol components, culminating in the Modring abstraction for managing sub-protocol instances, which allows space-optimal buffering of incoming messages. Additionally, we developed and implemented several optimisations improving Alea-BFT and the performance of the Mir framework.

The results of our experimental evaluation substantiate our methodology and offer opportunities for improving the Mir framework and Alea-BFT. Firstly, they validate our principled approach to implementing distributed protocols and the resulting abstractions, refining the design of the Mir framework. Secondly, they provide key insights for tuning the developed Alea-BFT optimisations and proposing new ones. Thirdly, the comparison with the ISS-PBFT protocol strengthens the claim the performance of asynchronous BFT (particularly Alea-BFT) is on par with state-of-the-art partially-synchronous protocols, which is crucial for real-world applications. Fourthly, they show the faster reaction time of Alea-BFT to crashes when compared to the same state-of-the-art partially-synchronous protocol.

In summary, the two main conclusions from this work are the importance of implementing systems with resource usage in mind and the readiness of Alea-BFT for real-world usage.

6.2 Future Work

During our research, we focused on implementing and evaluating Alea-BFT in the context of a real-world application – Filecoin subnets – aiming to foster the adoption of Alea-BFT and asynchronous BFT protocols in general. Throughout this process, we identified the strengths and limitations of our current approach and proposed directions for further research and development. Below, we summarise our main ideas.

Firstly, we implemented Alea-BFT with real-world usage in mind but left distributed threshold key generation and online system membership changes as future work due to their complexity.

Secondly, while the Modring successfully restricts sub-module resource usage by carefully choosing which sub-modules are live at any given time, it is fundamentally at odds with the existing Mir network stack, which reliably delivers all messages. To drop incoming messages, we developed SnoozeNet to track message reception and re-transmit them as required. However, this module is complex and interacts poorly with the network module's outgoing message queue. Thus, we propose researching a different approach to the network stack focused on sub-module-scoped communication channels and queueing.

Thirdly, our evaluation of Alea-BFT shows competitive performance with state-of-the-art partially-synchronous BFT protocols but only considers well-behaved networks and crash faults. More research is needed to evaluate the impact of adversarial networks and Byzantine faults on these systems.

Fourthly, the Mir framework's reusable modular components facilitate protocol implementation, but the design of Mir itself complicates this process. Concretely, Mir's event-based architecture requires defining events for all intermediate operations and various context structures to pass between operations, straining productivity during early exploration. Additionally, intra-module concurrency is impossible without breaking simulation-based testing due to introducing non-determinism (e.g. Goroutine Pools from Section 4.6.4). Furthermore, it complicates the use of traditional debugging and (distributed) tracing tools such as *gdb* and *Jaeger*. Therefore, we propose exploring alternative designs, favouring function calls and limiting event stream usage to appropriate situations while maintaining Mir's reproducible execution guarantees that enable simulation-based testing. Preliminary research suggests we can use Rust's Futures to allow intra-module concurrency and still be able to conduct simulations with a specially crafted runtime.

Bibliography

[1] D. S. Antunes, A. N. Oliveira, A. Breda, M. G. Franco, H. Moniz, and R. Rodrigues, “Alea-BFT: Practical asynchronous byzantine fault tolerance,” in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 313–328. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/antunes

[2] A. De la Rocha, L. Kokoris-Kogias, J. M. Soares, and M. Vukolić, “Hierarchical consensus: A horizontal scaling framework for blockchains,” in *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 2022, pp. 45–52.

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

[4] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of distributed consensus with one faulty process,” in *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, ser. PODS ’83. New York, NY, USA: Association for Computing Machinery, 1983, pp. 1–7. [Online]. Available: https://doi.org/10.1145/588058.588060

[5] C. Protocol Labs, “Mir - the distributed protocol implementation framework,” Accessed: December 2023. [Online]. Available: https://github.com/consensus-shipyard/mir/blob/e100175138a4fd8947b6757452334698ee518967/README.md

[6] C. Stathakopoulou, M. Pavlovic, and M. Vukolić, “State machine replication scalability made simple,” in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 17–33.

[7] M. Pavlovic, “Trantor: Modular state machine replication,” September 2023, Accessed: December 2023. [Online]. Available: https://github.com/consensus-shipyard/trantor-doc/blob/47dfc316a6d81604e1c567b823358f53fdfde4b4/main.pdf

[8] “Lotus consensus interface,” Accessed: November 2022. [Online]. Available: https://github.com/filecoin-project/eudico/blob/063b8aea67adf2c853631f6b6ec4dbb9249e78aa/chain/consensus/iface.go

[9] D. Schwartz, N. Youngs, and A. Britto, “The ripple protocol consensus algorithm,” 2014, Accessed: May 2024. [Online]. Available: https://ripple.com/files/ripple_consensus_whitepaper.pdf

[10] M. Travis, “Ripple: the most (demonstrably) scalable blockchain,” Oct. 2017, Accessed: May 2024. [Online]. Available: https://highscalability.com/ripple-the-most-demonstrably-scalable-blockchain/

[11] O. Nordström and C. Dovrolis, “Beware of BGP attacks,” *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 2, pp. 1–8, apr 2004. [Online]. Available: https://doi.org/10.1145/997150.997152

[12] M. Ohring and L. Kasprzak, “Chapter 7 - environmental damage to electronic products,” in *Reliability and Failure of Electronic Materials and Devices*, 2nd ed., M. Ohring and L. Kasprzak, Eds. Boston: Academic Press, 2015, pp. 387–441. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780120885749000070

[13] M. Castro, B. Liskov *et al.*, “Practical byzantine fault tolerance,” in *OsDI*, vol. 99, no. 1999, 1999, pp. 173–186.

[14] E. Buchman, “Tendermint: Byzantine fault tolerance in the age of blockchains,” Ph.D. dissertation, University of Guelph, 2016.

[15] T. Crain, C. Natoli, and V. Gramoli, “Red Belly: A secure, fair and scalable open blockchain,” in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 466–483.

[16] K. Antoniadis, J. Benhaim, A. Desjardins, E. Poroma, V. Gramoli, R. Guerraoui, G. Voron, and I. Zablotchi, “Leaderless consensus,” *Journal of Parallel and Distributed Computing*, vol. 176, pp. 95–113, 2023.

[17] 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*. IEEE, 2009, pp. 135–144.

[18] Y. Amir, B. Coan, J. Kirsch, and J. Lane, “Prime: Byzantine replication under attack,” *IEEE transactions on dependable and secure computing*, vol. 8, no. 4, pp. 564–577, 2010.

[19] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, “The honey badger of BFT protocols,” in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 31–42.

[20] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[21] M. Ben-Or, “Another advantage of free choice (extended abstract) completely asynchronous agreement protocols,” in *Proceedings of the second annual ACM symposium on Principles of distributed computing*. New York, NY, USA: Association for Computing Machinery, 1983, pp. 27–30.

[22] G. Bracha, “An asynchronous [(n-1)/3]-resilient consensus protocol,” in *Proceedings of the third annual ACM symposium on Principles of distributed computing*, 1984, pp. 154–162.

[23] 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.

[24] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, “Speeding Dumbo: Pushing asynchronous BFT closer to practice,” in *Proceedings of the 29th Network and Distributed System Security Symposium (NDSS’22)*, 2022.

[25] Y. Gao, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, “Dumbo-NG: Fast asynchronous BFT consensus with throughput-oblivious latency,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 1187–1201.

[26] Z. Milosevic, M. Biely, and A. Schiper, “Bounded delay in byzantine-tolerant state machine replication,” in *2013 IEEE 32nd International Symposium on Reliable Distributed Systems*. IEEE, 2013, pp. 61–70.

[27] C. Stathakopoulou, D. Tudor, M. Pavlovic, and M. Vukolić, “Mir-BFT: Scalable and robust BFT for decentralized networks,” *Journal of Systems Research*, vol. 2, no. 1, 2022.

[28] Y. Desmedt, “Threshold cryptosystems,” in *International Workshop on the Theory and Application of Cryptographic Techniques*. Springer, 1992, pp. 1–14.

[29] C. Cachin, K. Kursawe, and V. Shoup, “Random oracles in constantipole: practical asynchronous byzantine agreement using cryptography,” in *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, 2000, pp. 123–132.

[30] J.-P. Aumasson, A. Hamelink, and O. Shlomovits, “A survey of ECDSA threshold signing,” *Cryptology ePrint Archive*, 2020.

[31] S. Das, T. Yurek, Z. Xiang, A. Miller, L. Kokoris-Kogias, and L. Ren, “Practical asynchronous distributed key generation,” in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 2518–2534.

[32] E. Kokoris Kogias, D. Malkhi, and A. Spiegelman, “Asynchronous distributed key generation for computationally-secure randomness, consensus, and threshold signatures.” in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1751–1767.

[33] A. Kate, Y. Huang, and I. Goldberg, "Distributed key generation in the wild," *Cryptology ePrint Archive*, 2012.

[34] 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.

[35] A. Mostéfaoui, H. Moumen, and M. 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*, 2014, pp. 2–9.

[36] A. Miller, "Bug in ABA protocol's use of common coin," November 2019, GitHub Pull Request. Accessed: December 2023. [Online]. Available: https://github.com/amiller/HoneyBadgerBFT/issues/59

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

[38] M. Vukolić, "The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication," in *International workshop on open problems in network security*. Springer, 2015, pp. 112–125.

[39] J. R. Douceur, "The sybil attack," in *International workshop on peer-to-peer systems*. Springer, 2002, pp. 251–260.

[40] The Rust Team, "The Rust programming language," Accessed: March 2024. [Online]. Available: https://www.rust-lang.org/

[41] N. D. Matsakis and F. S. Klock, "The Rust language," *ACM SIGAda Ada Letters*, vol. 34, no. 3, pp. 103–104, 2014.

[42] "BLS12-381 threshold signature library used by DRand," Accessed: February 2024. [Online]. Available: https://github.com/drand/kyber-bls12381

[43] S. Mitsunari, "Precompiled ETH2.0 BLS threshold signature library with bindings for golang," Accessed: February 2024. [Online]. Available: https://github.com/herumi/bls-eth-go-binary

[44] A. Breda, "Implement threshold cryptography," September 2022, GitHub Pull Request merged into the Mir codebase. Accessed: February 2024. [Online]. Available: https://github.com/consensus-shipyard/mir/pull/219

[45] R. Oliveira, R. Guerraoui, and A. Schiper, "Stubborn communication channels," Départment d'Informatique, Ecole Polytechnique Fédérale de Lausanne, Tech. Rep., 1998.

[46] E. Barker and J. Kelsey, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Special Publication (NIST SP), National Institute of Standards

and Technology, Gaithersburg, MD, 2015. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-90Ar1

[47] A. Breda, “Set of scripts for running Mir on the RNL Cluster,” Accessed: May 2024. [Online]. Available: https://github.com/abread/mir-slurm

A

Alea-BFT Orchestrator Internals

Our work implements Alea-BFT in Mir/Trantor, which entails creating an orchestrator module that coordinates the interactions of Alea-BFT's components with Trantor components and the replicated application. To manage the complexity of this component, we opted to subdivide into duties which are only briefly presented in Section 3.7.1.

In this appendix we describe in detail a Queue Selection Policy abstraction that aids coordination between duties and the duties themselves.

Note that the orchestrator, like many Mir modules, processes events sequentially. Therefore, despite the existence of multiple upon rules for the same events, no two upon rules run concurrently. Instead, they run by order of appearance in the module's code, which corresponds to the order of appearance in this document.

A.1 Queue Selection Policy

Alea-BFT selects a queue for each agreement round based on a deterministic policy. To allow for different policies and enable checkpointing (which must include the state of the queues), we abstracted queue selection as a Golang interface, `QueueSelectionPolicy`, shown in Listing A.1. This interface was modelled after the existing `LeaderSelectionPolicy` interface in Trantor/ISS, which selects the designated leader node for each ISS sequenced broadcast instance (referred to as orderers in Trantor).

In this work, we only implement a round-robin policy, which selects the next queue in a round-robin fashion. Additionally, the optimisation described in Section 4.2 extends this interface to allow mapping a slot to the next agreement round that decides its delivery.

```
type Slot struct {
    QueueID uint32
    QueueSlot  uint64
}

interface QueueSelectionPolicy {
    // Map agreement round/sequence number to a slot, if the mapping is known.
    // Mapping must be known for the next agreement round.
    // Returns the slot corresponding to the given sequence number and whether
    // the mapping is known.
    Slot(sn uint64) (Slot, bool)

    // Update the state of the queue selection policy with the result
    // of an agreement round.
    // This method can only be called exactly once for each agreement round
    // and in order.
    // Returns the slot that is delivered in this sequence number (if any)
    // and whether a slot is delivered.
    DeliverSn(sn uint64, agDecision bool) (Slot, bool)

    // Number of the next sequence number to be delivered (next agreement round number).
    NextSn() uint64

    // Next queue slot to be delivered in a given queue
    QueueHead(queueID uint32) uint64

    // Serialises the state of the queue selection policy (for checkpointing purposes).
    Bytes() ([]byte, error)
}

// Deserialises a queue selection policy from a byte slice.
function QueueSelectionPolicyFromBytes([]byte) (QueueSelectionPolicy, error)
```

Listing A.1: Queue Selection Policy Interface

A.2 Batch Cutting Duty

The orchestrator component controls when batches are created and disseminated. A naive approach would continuously instruct the broadcast component to create and disseminate new batches. However, broadcasting batches without accounting for the comparatively slower agreement leads to unbounded resource usage. Thus, the orchestrator component strives to maintain a limited backlog of unagreed batches. This allows the broadcast component to accept a limited window of incoming VCB instances – `MaxOwnUnagreedBatches`.

We will now describe the concrete implementation of this control, which is presented as pseudocode in Listing A.2. This part of the orchestrator tracks the current queue selection policy (qsp, managed by the agreement loop control duty described in Appendix A.3), the ID of the next slot to be requested ($next$), and whether the broadcast component is currently working on a new batch ($stalled$).

The first upon rule (Lines 6 to 8) is triggered when a new availability certificate is delivered pertaining to the local node's queue. It increments the next slot ID (Line 7) and registers the batch-cutting process as stalled (Line 8), allowing the second upon rule to request a new batch.

The second upon rule (Lines 9 to 12), which runs after the orchestrator processes any set of events (crucially, including the `Init` event which is sent to all newly instantiated modules in Mir), instructs the broadcast component to create and disseminate a new batch (Line 11) if we are not waiting for it to do so already (i.e., $stalled$ is $True$) and if there are no more than `maxOwnUnagreedBatches` unagreed but already broadcast (Line 10).

```
 1: **shared state variables:**
 2:     $qsp$

 3: **local state variables:**
 4:     $next \leftarrow 0$
 5:     $stalled \leftarrow True$

 6: **upon** broadcast component delivering slot $s = \langle ownQueue, next \rangle$ **do**
 7:     $next \leftarrow next + 1$
 8:     $stalled \leftarrow True$

 9: **upon** finished processing batch of events in orchestrator **do**
10:     **if** $stalled \wedge next - qsp.QueueHead(ownQueue) < maxOwnUnagreedBatches$ **then**
11:         **send event** $\langle RequestCert, next \rangle$ to broadcast component
12:         $stalled \leftarrow False$
```

Listing A.2: Alea-BFT Orchestrator - Batch Cutting

A.3 Agreement Loop Control Duty

Alea-BFT describes an agreement loop, which decides on the delivery of a head of a queue in each iteration. In the original description of Alea-BFT, this loop resides in the agreement component. However, this implementation moved it to the orchestrator component and uses the agreement component as a facade for executing agreement rounds in series. Listing A.3 presents the pseudocode for the agreement loop control, which we will now describe.

The agreement loop control is responsible for providing inputs and processing the outputs of agreement rounds. To this end, it tracks the current agreement round number (r), whether the agreement component is stalled for input ($agStalled$), the queue selection policy (qsp), and a set of unagreed slots that were already delivered by the broadcast component ($unagreedSlots$). Of these state variables, only $unagreedSlots$ is local to the agreement loop control, while the others are shared with the rest of the orchestrator component.

The first upon rule (Lines 6 to 7) adds slots delivered by the broadcast component to the set of unagreed slots.

The second upon rule (Lines 8 to 17) processes the output of an agreement round. Note that, despite agreement rounds being sequential in regular operation, the slow node recovery mechanism may fast-forward the r variable to a higher value (right past the restored checkpoint). Therefore, this upon rule ignores delivery events for older rounds (only the delivery of round r is processed). If the decision is to deliver, we do so (Lines 11 to 12), removing it from the set of unagreed slots (Line 13) and instructing the broadcast component to free the corresponding VCB instance (Line 14). Note that freeing the VCB instance does not free the batch nor its metadata: it is merely a back-pressure mechanism that bounds the number of unagreed slots broadcast by each node. Additionally, regardless of the result of the agreement round, the queue selection policy is updated (Line 15), the agreement round number is incremented (Line 16), and the agreement component is marked as stalled (Line 17).

Finally, the third upon rule (Lines 18 to 23) provides input for the next agreement round, starting it if not already running. To this end, it determines the corresponding slot for the current agreement round (Line 18), computes the input for the agreement round by checking if the corresponding slot is locally available (Line 21), sends the input to the agreement component (Line 22), and marks the agreement component as no longer stalled (Line 23). This upon rule could be merged into the second upon rule, providing input for the next agreement round when the current round finishes. However, separating these rules facilitates the optimisation described in Section 4.3.

```
1:  **shared state variables:**
2:      $r \leftarrow 0$
3:      $agStalled \leftarrow True$
4:      $qsp$
5:      $unagreedSlots \leftarrow \{\}$

6:  **upon** broadcast component delivering slot $s$ **do**
7:      $unagreedSlots \leftarrow unagreedSlots \cup \{s\}$

8:  **upon** agreement component delivering round $r$ with decision $d$ **do**
9:      $s_r \leftarrow qsp.Slot(r)$
10:     **if** $d == 1$ **then**
11:         $cert \leftarrow \langle ALEA-CERT, s\rangle$
12:         **send event** $\langle DeliverCert, s\rangle$ to batch fetcher
13:         $unagreedSlots \leftarrow unagreedSlots \setminus \{s_r\}$
14:         **send event** $\langle FreeSlot, s_r\rangle$ to broadcast component
15:     $qsp.DeliverSn(r, d)$
16:     $r \leftarrow r + 1$
17:     $agStalled \leftarrow True$

18: **upon** finished processing batch of events in orchestrator **do**
19:     **if** $agStalled$ **then**
20:         $s_r \leftarrow qsp.Slot(r)$
21:         $input \leftarrow True$ if $s_r \in unagreedSlots$ else $False$
22:         **send event** $\langle InputValue, r, input\rangle$ to agreement component
23:         $agStalled \leftarrow False$
```

Listing A.3: Alea-BFT Orchestrator - Agreement Loop Control

A.4 Checkpointing And Garbage Collection Duty

The orchestrator creates system checkpoints every `EpochLength` agreement rounds summarising replicated state machine application state and protocol state. As checkpoints become ”stable” (i.e., are certified by enough nodes), the node can free resources associated with older checkpoints, clearing all state associated with epochs before the latest checkpointed epoch plus `RetainedEpochCount`. This process is essential for limiting the resource usage of Alea-BFT, as it allows the system to free resources that are no longer needed. Listing A.4 presents the orchestrator's checkpointing logic, which we will now describe.

The checkpointing logic shares three variables with the remaining orchestrator parts: r (current agreement round number), qsp (queue selection policy), and $stableCheckpoint$ (latest stable checkpoint). r and qsp are maintained by the agreement loop control logic, but $stableCheckpoint$ is updated by the checkpointing logic.

The first upon rule (Lines 4 to 7) is triggered by the delivery of an agreement round (excluding out-of-order deliveries caused by the slow node recovery procedure). Note that variable r refers to the round number of the next agreement round to be delivered, as it is incremented by the agreement loop control logic, which runs before this upon rule. Thus, we will always observe the round delivery event for round $r' = r - 1$. If the delivered round is the last round of an epoch, the orchestrator starts checkpointing the current epoch and advances to the next using the CHECKPOINT-ADVANCE-EPOCH procedure.

The second upon rule (Lines 8 to 10) is triggered by the delivery of a checkpoint certificate, marking the end of the checkpointing process for epoch $epoch$. If the new stable checkpoint is newer than the saved latest stable checkpoint (Line 9), the orchestrator saves it (Line 10) using the SAVE-LATEST-CHECKPOINT procedure, which will be described in the remainder of this section.

The CHECKPOINT-ADVANCE-EPOCH procedure (Lines 11 to 16) advances the epoch using the ADVANCE-EPOCH procedure (Lines 17 to 20), which conveys the new epoch to the broadcast component, agreement component, and batch fetcher. Afterwards, CHECKPOINT-ADVANCE-EPOCH serialises the queue selection policy (Line 13) and starts a new checkpointing module instance (Lines 14 and 15). However, the checkpointing module must eventually receive a snapshot of the application state. Therefore, this procedure requests a snapshot of the application state from the batch fetcher and instructs the batch fetcher to send this snapshot to the corresponding checkpointing instance (Line 16). Note that the snapshot request is sent to the batch fetcher and not directly to the application, ensuring that the snapshot is taken before any new batches are delivered.

The SAVE-LATEST-CHECKPOINT procedure (Lines 21 to 27) processes a new latest checkpoint, storing it to allow recovery of other nodes and garbage-collecting information from older checkpoints. Firstly, it stores the new stable checkpoint (Line 22). Secondly, it computes the new retention index for the checkpoint (Line 23), which is the epoch number of the checkpoint minus the number of retained epochs. Finally, if the new retention index is positive, there is data to clear. Thus, we instruct Alea-BFT's components to free resources associated with epochs before the new retention index (Lines 25 and 26). Furthermore, it frees all checkpointing instances associated with epochs before the new retention index (Line 27).

```
shared state variables:
    $r$, $qsp$
    $stableCheckpoint \leftarrow \bot$

upon agreement component delivering round $r' = r - 1$ with decision $d$ do    ▷ Global variable $r$ was already incremented by agreement loop control (Line 16 in Listing A.3), so here we will always observe the round delivery event for round $r - 1$.
    if $r' \% EpochLength = EpochLength - 1 \wedge r' > 0$ then
        $newEpoch \leftarrow \lfloor \frac{r}{EpochLength} \rfloor$
        CHECKPOINT-ADVANCE-EPOCH($newEpoch$)

upon checkpoint module $e$ delivering checkpoint certificate $chkp = \langle$CHKP, $epoch, qspData, state\rangle$ do
    if $stableCheckpoint = \bot \vee epoch > stableCheckpoint.epoch$ then
        SAVE-LATEST-CHECKPOINT($chkp$)

procedure CHECKPOINT-ADVANCE-EPOCH($newEpoch$)
    ADVANCE-EPOCH($newEpoch$)
    $serialisedPolicy \leftarrow qsp.Bytes()$
    $chkpModID \leftarrow \langle$thresh-checkpoint, $newEpoch\rangle$
    send event $\langle$NewInstance, $newEpoch, serialisedPolicy\rangle$ to thresh-checkpoint factory module
    send event $\langle$AppRequestSnapshot, $chkpModID\rangle$ to batch fetcher

procedure ADVANCE-EPOCH($newEpoch$)
    send event $\langle$NewEpoch, $newEpoch\rangle$ to agreement component
    send event $\langle$NewEpoch, $newEpoch\rangle$ to broadcast component
    send event $\langle$NewEpoch, $newEpoch\rangle$ to batch fetcher

procedure SAVE-LATEST-CHECKPOINT($chkp$)
    $stableCheckpoint \leftarrow chkp$
    $firstRetEpoch \leftarrow chkp.epoch - RetainedEpochCount$
    if $firstRetEpoch > 0$ then
        send event $\langle$GarbageCollect, $firstRetEpoch\rangle$ to agreement component
        send event $\langle$GarbageCollect, $firstRetEpoch\rangle$ to broadcast component
        send event $\langle$GarbageCollect, $firstRetEpoch\rangle$ to thresh-checkpoint factory module
```

Listing A.4: Alea-BFT Orchestrator - Checkpointing

A.5 Slow Node Detection And Recovery Duty

Alea-BFT's orchestrator component is responsible for detecting slow nodes and recovering them. To this end, it tracks the most recent epoch observed by each remote node and sends state transfer messages to nodes that are too far behind. Concretely, we consider a node too far behind the rest of the system if the last observed epoch is lower than the current epoch minus `RetainedEpochCount`. This condition selects nodes behind the retained epoch range, meaning they are likely unable to make progress without a state transfer message.

We will now present the logic associated with this task, referring to its pseudocode in Listing A.5. This orchestrator duty requires access to all shared state, allowing it to fast-forward the orchestrator to a future epoch from a checkpoint. Additionally, it maintains a single local state variable – $nodeEpochs$ –

which tracks the most recent epoch that the local node observed on a remote node.

The first upon rule (Lines 5 to 8) is triggered by an internal `HelpNode` event, which signals the orchestrator that some remote node is likely too far behind the rest of the system and needs help to catch up. Thus, the orchestrator sends the node requiring help a state transfer messaged derived from the latest checkpoint (Line 8). However, HelpNode events may be emitted by multiple components in duplicate. Thus, the orchestrator will only send the state transfer message if the last observed epoch in the node is outside of the nominal range (Line 6) and will register the epoch of the checkpoint in the sent state transfer message as the latest observed epoch in the node being helped (Line 7).

The second upon rule (Lines 9 to 11) is triggered when a remote node p_i signs a checkpoint for round e (corresponds to an `EpochProgress` event from the `thresh-checkpointing` module). If a correct node signs a checkpoint, it must have processed all protocol messages up to the corresponding epoch. Thus, if e is newer than the previously registered last observed epoch for node p_i (Line 10), we update this record accordingly (Line 11).

The third upon rule (Lines 12 to 17) is triggered by the receipt of a state transfer message from a remote node p_i. We begin by updating the last observed epoch for node p_i (Lines 13 and 14) similarly to the second upon rule (Lines 10 and 11). Afterwards, if the local node is too far behind the checkpoint (Lines 15 and 16), it will instruct the checkpoint certificate validator to validate it (Line 17).

The fourth upon rule (Lines 18 to 31) completes the recovery process and is triggered by a successful validation[1] of a checkpoint certificate. Similarly to the former upon rule, it checks if the local node is too far behind the checkpoint (Lines 19 and 20) and, if so, instructs the batch fetcher, agreement component, and broadcast component to restore their states to the checkpointed state (Lines 21 to 23). Note that, once again, the batch fetcher serves as a proxy for the application, ensuring that the restoration process is serialised with the delivery of new batches. Additionally, it restores the state of the orchestrator module itself from the checkpoint. Concretely, it deserialises and updates the queue selection policy (Line 24), updates the current agreement round number (Line 25). This leaves the orchestrator in the beginning of the first round of the epoch after the one restored from the checkpoint. Thus, we mark the agreement component as stalled (Line 26), and advance to that epoch (Line 27). Furthermore, the $unagreedSlots$ set is cleared of all already delivered slots (Lines 28 to 30). Finally, it saves the restored checkpoint as the latest stable checkpoint (Line 31).

[1] Invalid checkpoints could be used to build failure detectors, but we chose to leave that as future work.

```
shared state variables:
    $stableCheckpoint$, $r$, $qsp$, $unagreedSlots$, $agStalled$

local state variables:
    $nodeEpoch[p_i] \leftarrow 0$ for each node $p_i$, $i \in 0, \dots, N-1$

upon event ⟨HelpNode, $p_i$⟩ do
    if $nodeEpoch[p_i] < stableCheckpoint.epoch$ then
        $nodeEpoch[p_i] \leftarrow stableCheckpoint.epoch$
        send event ⟨SendMessage, $p_i$, ⟨RECOVER, $stableCheckpoint$⟩⟩ to network

upon node $p_i$ signing checkpoint for round $e$ do  ▷ EpochProgress event from thresh-checkpointing/$e$ module
    if $e > nodeEpoch[p_i]$ then
        $nodeEpoch[p_i] \leftarrow e$

upon receiving message ⟨RECOVER, $chkp$ = ⟨CHKP, $chkpEpoch, qspData, state$⟩⟩ from node $p_i$ do
    if $nodeEpoch[p_i] < chkpEpoch$ then
        $nodeEpoch[p_i] \leftarrow chkpEpoch$
    $ownEpoch \leftarrow \left\lceil \frac{r}{\text{EpochLength}} \right\rceil$
    if $chkpEpoch - \text{RetainedEpochCount} > ownEpoch$ then            ▷ check if own node is behind
        send event ⟨ValidateCheckpoint, $chkp$⟩ to thresh-checkpointing-validator

upon validating checkpoint certificate $chkp$ = ⟨CHKP, $chkpEpoch, qspData, state$⟩ do
    $ownEpoch \leftarrow \left\lceil \frac{r}{\text{EpochLength}} \right\rceil$
    if $chkpEpoch - \text{RetainedEpochCount} > ownEpoch$ then            ▷ check if own node is still behind
        send event ⟨RestoreState, $chkp$⟩ to batch fetcher
        send event ⟨RestoreState, $chkp$⟩ to agreement component
        send event ⟨RestoreState, $chkp$⟩ to broadcast component
        $qsp \leftarrow QueueSelectionPolicyFromBytes(qspData)$        ▷ error handler omitted for simplicity (valid checkpoints always have valid queue selection policies)
        $r \leftarrow qsp.NextSn()$
        $agStalled \leftarrow True$
        ADVANCE-EPOCH($chkpEpoch + 1$)
        for each $slot \in unagreedSlots$ do
            if $slot.QueueSlot < qsp.queue[slot.QueueID]$ then
                $unagreedSlots \leftarrow unagreedSlots \setminus \{slot\}$
        SAVE-LATEST-CHECKPOINT($chkp$)
```

Listing A.5: Alea-BFT Orchestrator - Slow Node Detection and Recovery

B

Full Experimental Configuration

In this appendix we present the full list of experimental configuration. Appendix B.1 includes the list of configuration that are used in both protocols. Appendix B.2 includes the list of configuration that are used in ISS-PBFT. Finally, Appendix B.3 includes the list of configuration that are used Alea-BFT.

B.1 Common Experimental Configuration

- *TxGen.PayloadSize* – Size of each transaction payload (in bytes): 256[1].
- *TxGen.NumClients* – Number of closed-loop clients co-located with node: depends on the experiment.
- *CryptoSeed* – seed for cryptographic key generation: pseudorandom, generated from hash of experimental configuration and repetition number.
- *RetainedEpochs* – Number of most-recent epochs that are kept during garbage-collection: 2.

[1] In practice, transactions are larger than this value because they also carry a client ID string and are serialised using protobuf

- *EpochLength* – Number of agreed batches in an epoch: 256[2].
- *Mempool.MaxTransactionsInBatch* – Maximum number of transactions per batch: depends on the experiment.
- *Net.MaxMessageSize* – Maximum message size: $max\{2MiB, c \times 1.05,$ *Mempool.MaxTransactionsInBatch* $\times$ *TxGen.PayloadSize* $\times 1.05\}$, where c is the approximated size of a checkpoint message.
- *Net.ConnectionBufferSize* – Per-destination outgoing message queue maximum capacity: 2048.
- *Net.MaxDataPerWrite* – Maximum size of chunk of data written to connection (in a single write syscall): $100kiB$.

B.2 ISS-PBFT Experimental Configuration

- *SegmentLength* – Length of an ISS segment in sequence numbers: $max\{\lfloor$*EpochLength*$/N\rfloor, 2\}$ where N is the number of nodes in the system
- *MaxProposeDelay* – Maximum time between two proposals of a leader: $4s$
- *MsgBufCapacity* – Buffer capacity in factory module instances: $32MiB$
- *CatchUpTimerPeriod* – Length of time between the local node checking if other nodes have fallen behind: *MaxProposeDelay*
- *CheckpointResendPeriod* – Re-transmission interval for checkpoint messages: *MaxProposeDelay*
- *LeaderSelectionPolicy* – Type of leader selection policy: Blacklist (excludes the nodes that most recently timed out from the set of leaders, limited to F excluded nodes)
- *PBFTDoneResendPeriod*: *MaxProposeDelay*
- *PBFTCatchUpDelay*: *MaxProposeDelay*
- *PBFTViewChangeSNTimeout*: $4 \times$ *MaxProposeDelay*
- *PBFTViewChangeSegmentTimeout*: $4 \times$ *SegmentLength* $\times$ *MaxProposeDelay*
- *PBFTViewChangeResendPeriod*: *MaxProposeDelay*
- *Mempool.MinTransactionsInBatch* – Minimum number of transactions per batch: 0
- *Mempool.BatchTimeout* – Time to wait for a batch to fill up: *MaxProposeDelay*

[2]The ISS implementation does not currently support setting the epoch length directly. The actual epoch length used in ISS experiments is $max\{\lfloor$*EpochLength*$/N\rfloor, 2\} \times N'$, where N is the number of nodes in the system and N' the number of leaders selected by the leader selection policy.

B.3 Alea-BFT Experimental Configuration

- *InstanceUID* – a unique identifier for an instance of Alea-BFT used for preventing replay attacks: pseudorandom, generated from hash of experimental configuration and repetition number
- *MaxConcurrentVcbPerQueue* – Size of Modring of VCB instances in each Alea-BFT queue: $max\{(\textit{EpochSz}/N+1)\times 2, \textit{MaxOwnUnagreedBatches}\times 2\}$
- *MaxOwnUnagreedBatches* – Maximum number of batches that are mid-broadcast or already broadcast but not delivered in Alea-BFT: 2
- *MaxAbbaRoundLookahead* – Size of Modring of ABBA round instances in each ABBA instance: 4
- *MaxAgRoundLookahead* – Size of Modring of ABBA instances in Alea-BFT's agreement component: $\textit{EpochLength}\times 2$
- *MaxAgRoundEagerInput* – Maximum number of agreement rounds that can have eager input at any given time: $\textit{EpochLength}-1$
- *EstimateWindowSize* – Duration estimate window size: 64
- *MaxExtSlowdownFactor* – Factor that controls how slower the F slowest nodes can be in relation to the rest of the system for duration estimation purposes (1 means both are the same speed): 1.5
- *QueueSelectionPolicyType* – Type of Queue Selection Policy: Round-Robin
- *MaxAgStall* – Maximum agreement 0-input delay: $10s$
- *Mempool.MinTransactionsInBatch* – Minimum number of transactions per batch: 1
- *Mempool.BatchTimeout* – Time to wait for a batch to fill up: 0
- *SnoozeNet.RetransmissionLoopInterval* – Length of time between message re-transmissions: $5s$
- *SnoozeNet.MaxRetransmissionBurst* – Maximum number of re-transmitted messages at a time: 64