
# APBFT

## Adapting the PBFT Consensus Algorithm to the Asymmetric Trust Setting

## Master Thesis

Bernhard Jonathan

Universität Freiburg

August 2025

Prof. Dr. Christian Cachin
Juan Villacis
Michael Senn

Cryptology and Data Security Group
Institute of Computer Science
University of Bern, Switzerland



# Abstract

Consensus protocols are fundamental to distributed computing, enabling processes to agree on a single value even in the presence of faults. Traditional consensus mechanisms, such as Paxos, PBFT, HotStuff, and Tendermint, assume a symmetric trust model where all nodes share uniform assumptions about fault tolerance. While effective in controlled environments, this assumption often limits applicability in large-scale, decentralized, or geo-distributed networks, where nodes may hold heterogeneous trust views.

This thesis investigates consensus under asymmetric trust, where each node specifies its own fail-prone sets. We present the first empirical evaluation of two consensus paradigms adapted to this setting: a leader-driven PBFT variant (PAPBFT) and Asymmetric Randomized Consensus, a fully asynchronous randomized consensus protocol. Our experiments demonstrate that PAPBFT achieves low latency in fault-free scenarios but is sensitive to faulty leaders, whereas the randomized protocol maintains stable performance regardless of faults.

To make such protocols practical, we examine the computational complexity of key quorum-related tasks and prove that enumerating the tolerated system of an asymmetric quorum system, which a key component of several algorithms, is NP-hard. We address this challenge by introducing the superset recognizer abstraction, and show that an efficient recognizer for a quorum system can be transformed into recognizers for both its kernel system and its tolerated system. This allows the algorithms to be implemented entirely on top of quorum superset recognizers, avoiding the need to explicitly enumerate these complex structures.

Building on prior work, we extend the PBFT algorithm with a leader-change primitive tailored for asymmetric trust and propose a novel asymmetric variant of the PBFT consensus mechanism that manages unequal role assignments among nodes. We provide a formal correctness proof of our PBFT variant, ensuring both safety and liveness in asymmetric trust environments. These contributions establish a practical foundation for reliable and efficient consensus in heterogeneous distributed systems.

# Contents

# 1 Introduction

The consensus problem is a core abstraction in distributed computing, requiring processes to agree on a single value among those proposed, even in the presence of faults. It underpins systems that require consistent transaction ordering, as repeated instances can be used to build a global sequence of decisions. Protocols such as PAXOS [14], PBFT [7], HOTSTUFF [22], and TENDERMINT [5] are well-established consensus mechanisms in practice, forming the foundation of many replicated services and permissioned blockchain systems.

These protocols are built on top of the *symmetric trust model*, where all nodes share a uniform assumption about the number and type of faults the system must tolerate. Typically, this is captured by a fixed threshold, e.g., the assumption that at most $f$ out of $n$ nodes are Byzantine, meaning they may behave in arbitrary or malicious ways. Under this assumption, quorum systems are constructed such that every quorum has sufficient overlap with others to guarantee both *safety* and *liveness* properties.

This threshold-based model can be more generally captured by the framework of *Byzantine quorum systems (BQS)* [17]. Instead of relying on numerical thresholds, BQS define a set of *fail-prone sets*: collections of nodes that might fail together. A quorum system is then a set of node subsets that intersect with one another and contain a sufficient number of correct processes. In the symmetric BQS model, all nodes share the same quorums and fail-prone sets, preserving the assumption of a homogeneous trust structure.

However, this symmetric trust assumption can become a significant limitation in real-world systems. In large-scale, geo-distributed, or decentralized networks, participants may have very different views of which nodes they consider trustworthy. Nodes operated by independent entities often cannot agree on a global fault assumption.

To address these limitations, BQS have been extended to support *asymmetric trust*. In this setting, as introduced by Damgård et al. [8] and later refined by Alpos et al. [1], each node specifies its own view of fail-prone sets, reflecting its local trust assumptions.

Zanolini [23] introduced two consensus algorithms for the asymmetric trust setting: a variant of the PBFT algorithm as proposed by Cachin et al. [6], and a randomized consensus algorithm based on the work of Mostéfaoui et al. [19]. However, Zanolini neither provides an implementation of the proposed algorithms nor specifies the full PBFT mechanism, leaving, in particular, the liveness mechanism open. Building on this foundation, we complete Zanolini's PBFT variant with a leader-change primitive tailored to the asymmetric trust setting, enabling its practical execution. We implement both of Zanolini's algorithms and empirically evaluate their performance. Furthermore, we propose a novel asymmetric variant of PBFT and provide a formal proof of its correctness.

The remainder of this thesis is structured as follows.

- **Chapter 2** discusses related work, particularly other consensus protocols that address asymmetric trust.
- **Chapter 3** presents the preliminaries, introducing the system model and quorum systems.
- **Chapter 4** details hardness results related to the computation of guilds in asymmetric quorum systems and discusses how to efficiently operationalize the reviewed quorum-based algorithms.
- **Chapter 5** reviews the Asymmetric Randomized Consensus algorithm.
- **Chapter 6** examines an asymmetric version of PBFT and introduces a novel variation of the protocol.

- **Chapter 7** presents the empirical evaluation of the two reviewed algorithms.
- **Chapter 8** outlines future work, discussing potential extensions and open questions.
- **Chapter 9** concludes the thesis, summarizing the results.

# 2
# Related Work

The study of asymmetric trust in distributed consensus has been formalized through several foundational works. Damgård et al. [8] were the first to introduce the notion of asymmetric trust, laying the theoretical groundwork for subsequent research. Building on this foundation, Alpos et al. [1] formalized asymmetric trust as a generalization of symmetric Byzantine quorum systems [17], extending classical consistency and availability properties to a more flexible setting. In their framework, they also introduced the concept of a guild, a subset of correct nodes whose presence ensures system progress, and proved that their proposed consensus algorithms satisfy both safety and liveness under this model.

Subsequently, Li et al.[15] showed that consistency and availability alone are insufficient to guarantee consensus in asymmetric trust environments, emphasizing the necessity of the guild condition identified by Alpos et al. They also developed their own model of asymmetric Byzantine quorum systems, derived from the Federated Byzantine Agreement System (FBAS) used in Stellar [18]. In this formulation, quorum system properties are defined relative to a fixed set of predetermined Byzantine processes. By contrast, the Alpos et al.[1] approach defines these properties in terms of the fail-prone system introduced by Damgård et al. [8], which leads to a more natural expression of the consistency property. This formulation accommodates executions in which the quorums of correct processes, with incorrect trust assumptions, need not intersect in a correct process with the quorums of its peers. Such scenarios cannot be expressed in Li et al.'s [15] formalism, where the consistency property requires the quorums of all correct processes to intersect in at least one correct process.

Amores-Sesar et al. [3] present the first DAG-based asymmetric consensus protocol, which generalizes the renowned DAG-Rider [12] algorithm using the same formal framework adopted in this work. This work builds directly on the model developed by Alpos et al. [1].

In a somewhat independent line, Heterogeneous Paxos [21] generalizes Byzantine Paxos to settings with non-uniform trust. A further generalization of the protocol is that it allows learners to also operate under heterogeneous fault models. To capture these assumptions, the authors introduce the concept of a learner graph, where each learner is represented as a node labeled with its individual quorum system, the sets of validators it trusts to reach a decision. Edges between learners are labeled with safe sets, which specify the subsets of non-Byzantine nodes required to ensure that connected learners agree on their decision value.

Ripple [2, 20] and Stellar [18] are two of the most prominent blockchain systems built on asymmetric trust assumptions, enabling open-ended membership without requiring a globally agreed-upon validator set. Ripple requires each node to maintain a static *Unique Node List* (UNL), a list of trusted validators. A node only considers messages from nodes in its UNL and regards a transaction as validated if a sufficient proportion of those validators approve it. However, in practice, Ripple exhibits a relatively centralized structure: new participants are given a default UNL, which they can customize, but are discouraged from doing so, as poor choices could compromise the system's safety guarantees [18].

Stellar builds on and generalizes Ripple's approach by introducing the concept of a Federated Byzantine Agreement System (FBAS). In Stellar, each node defines one or more quorum slices, subsets of nodes it trusts to reach consensus. These slices collectively give rise to quorums, which are sets of nodes that contain a quorum slice for each of their members. The structure of quorum slices and quorums in Stellar resembles the quorum systems and guilds discussed earlier. However, the two differ fundamentally in their properties, as the notion of quorum

systems is grounded in the concept of fail-prone sets, whereas FBAS does not explicitly model the possible faulty processes in the system.

Both systems require careful configuration to ensure liveness: Ripple relies on timely communication and high UNL overlap [2], while Stellar requires that nodes form quorum slices with adequate intersection and availability [11]. However, Stellar also assumes some level of network synchrony to guarantee progress.

Cobalt [16], developed as a successor to Ripple's protocol, retains the asymmetric trust model in which each node maintains a locally defined trusted set (similar to UNLs), but improves resilience by reducing the required overlap between them and operating under full asynchrony, without assuming bounded message delays.

# 3
# Preliminaries

In this chapter, we introduce the system model and then review the notion of *Asymmetric Byzantine Quorum Systems*, originally proposed by Damgård et al. [8] and later extended by Alpos et al. [1]. These systems offer a mathematical framework for formalizing trust assumptions in distributed environments and form the basis of all the protocols discussed in this work.

To build toward this generalization, we first review the classical notion of *Symmetric Byzantine Quorum Systems*, introduced by Malkhi and Reiter [17], before moving on to the asymmetric setting.

## 3.1 System Model

**Processes.** In this thesis, our attention is restricted to *permissioned* systems of $n$ processes, denoted by $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$. The processes communicate with each other over a fully connected network.

**Executions and Faults.** An execution starts with all processes in a special initial state. The processes then repeatedly trigger events, as soon as their guards are satisfied, changing the processes' states. Every event executes as an atomic unit. Each event executed by some process is associated with a unique point in time $t \in \mathbb{R}$, inducing a total order on the events. However, the processes do not have access to a global clock and are therefore unaware of this global ordering.

A process that follows its protocol throughout its execution is called *correct*. Processes may *crash* during execution or *deviate arbitrarily* from their specification; such faulty processes are called *Byzantine*.

**Idealized Digital Signatures.** In the pseudocode, the cryptographic operation of generating digital signatures is abstracted into two operations, $\textit{sign}_i$ and $\textit{verify}_i$. Process $p_i$ invokes $\textit{sign}_i$ by providing a object $o$ as input and obtaining a signature $\sigma \in \{0,1\}^*$ as output. Only $p_i$ may invoke $\textit{sign}_i$. The operation $\textit{verify}_i$ may be called by any process: given an object $o$ and a signature $\sigma$, it returns TRUE if and only if $p_i$ previously invoked $\textit{sign}_i(o)$ and produced $\sigma$.

**Timing Assumptions.** We consider two different timing models for message delays:

- **Fully Asynchronous Model.** Messages may be delayed by an unbounded amount of time but are guaranteed to eventually be delivered.
- **Partially Synchronous Model [9].** The execution is divided into two phases. Before an unknown global stabilization time (*GST* $\in \mathbb{R}$), messages behave as in the fully asynchronous model. After *GST*, every message sent is delivered within at most $\Delta \in \mathbb{R}$ time units. Both *GST* and $\Delta$ are unknown to the processes, and no messages are lost.

## 3.2 Symmetric Byzantine Quorum System

Given a set of processes $\mathcal{P}$, a *fail-prone system* $\mathcal{F} \subseteq 2^{\mathcal{P}}$ encodes the trust assumptions underlying the Byzantine quorum system. Each set $F \in \mathcal{F}$ represents a group of processes that might fail together during the execution of a distributed protocol. In essence, $\mathcal{F}$ specifies the failure patterns that processes in $\mathcal{P}$ are designed to tolerate.

A *quorum system* $\mathcal{Q} \subseteq 2^{\mathcal{P}}$ is a collection of subsets of $\mathcal{P}$ that guarantees both safety and liveness, under the assumption that failures conform to the structure defined by $\mathcal{F}$.

The safety property, known as the **Consistency Property**, ensures that any two quorums $Q_1, Q_2 \in \mathcal{Q}$ intersect in at least one correct process. This overlap guarantees that when two processes make decisions based on quorum information, they do so with input from at least one common, correct source, an essential property for avoiding divergence and maintaining global consistency.

The liveness property, referred to as the **Availability Property**, ensures that for any admissible failure set $F \in \mathcal{F}$, there exists at least one quorum $Q \in \mathcal{Q}$ that contains only correct processes. This ensures that progress remains possible despite faults.

**Definition 1 (Symmetric Byzantine Quorum System).** Let $\mathcal{P}$ be a set of processes, and $\mathcal{F} \subseteq 2^{\mathcal{P}}$ a fail-prone system representing the possible sets of processes that may fail together.

A *Byzantine quorum system* for $\mathcal{F}$ is a collection of quorums $\mathcal{Q} \subseteq 2^{\mathcal{P}}$, where no quorum is contained in another, and each $Q \in \mathcal{Q}$ is called a quorum. The system satisfies the following two properties:

- **Consistency.** The intersection of any two quorums contains at least one correct process. Formally, for all $Q_1, Q_2 \in \mathcal{Q}$ and for all $F \in \mathcal{F}$,
$$Q_1 \cap Q_2 \nsubseteq F.$$
- **Availability.** For every fail-prone set $F \in \mathcal{F}$, there exists a quorum $Q \in \mathcal{Q}$ that is disjoint from $F$, i.e.,
$$\forall F \in \mathcal{F}, \exists Q \in \mathcal{Q} : Q \cap F = \emptyset.$$

The following definition introduces a conceptually simple yet powerful condition, known as the $Q^3$-condition, which determines whether a given fail-prone system $\mathcal{F}$ can be complemented by a quorum system $\mathcal{Q}$ that satisfies Definition 1.

**Definition 2 ($Q^3$-Condition).** A fail-prone system $\mathcal{F}$ satisfies the $Q^3$-*condition*, abbreviated as $Q^3(\mathcal{F})$, whenever it holds that
$$\forall F_1, F_2, F_3 \in \mathcal{F} : F_1 \cup F_2 \cup F_3 \neq \mathcal{P}.$$

Intuitively, the $Q^3$-condition states that no combination of three fail-prone sets can cover the entire set of processes. This condition captures a fundamental limit on the number of failures a quorum system can tolerate.

**Lemma 1 (Malkhi et al. [17]).** *Given a fail-prone system $\mathcal{F}$, a Byzantine quorum system for $\mathcal{F}$ exists if and only if $Q^3(\mathcal{F})$.*

**Core Sets.** A core set is another important concept when designing distributed protocols. It refers to a minimal set of processes that is guaranteed to include at least one correct process in every execution. The formal definition is as follows:

**Definition 3 (Core Set).** Let $\mathcal{P}$ be a set of processes and $\mathcal{F} \subseteq 2^{\mathcal{P}}$ a fail-prone system.

A *core set* for $\mathcal{F}$ is a set $C \subseteq \mathcal{P}$ satisfying:

- **Correctness Coverage.** The set $C$ contains at least one correct process in any failure scenario, i.e., for all $F \in \mathcal{F}$,
$$C \nsubseteq F \quad \text{or equivalently,} \quad C \cap (P \setminus F) \neq \emptyset.$$
- **Minimality.** No proper subset of $C$ satisfies the above property. That is,
$$\forall C' \subsetneq C, \exists F \in \mathcal{F} : C' \subseteq F.$$

We denote the collection of all core sets for $\mathcal{F}$ by $\mathcal{C}$.

**Kernels.** Lastly, we introduce the notion of a *kernel*, which generalizes the idea of a core set. A kernel is a minimal set of processes that intersects with every quorum in the system $\mathcal{Q}$. In this sense, every kernel includes at least one core set.

**Definition 4 (Kernel).** Let $\mathcal{P}$ be a set of processes, $\mathcal{F} \subseteq 2^{\mathcal{P}}$ a fail-prone system, and $\mathcal{Q} \subseteq 2^{\mathcal{P}}$ a quorum system. A *kernel* for $\mathcal{Q}$ is a set $K \subseteq \mathcal{P}$ such that:

- **Quorum Intersection.** The kernel intersects every quorum, i.e.,

$$\forall Q \in \mathcal{Q} : K \cap Q \neq \emptyset.$$

- **Minimality.** No proper subset of $K$ satisfies the above condition, i.e.,

$$\forall K' \subsetneq K, \exists Q \in \mathcal{Q} : K' \cap Q = \emptyset.$$

We denote the set of all kernels of the quorum system $\mathcal{Q}$ by $\mathcal{K}$, also referred to as the *kernel system*.

## 3.3 Asymmetric Byzantine Quorum System

The concept of a symmetric Byzantine quorum system can be generalized by allowing each process $p_i$ to specify its own individual fail-prone system $\mathcal{F}_i \subseteq 2^{\mathcal{P}}$, thereby expressing the potentially differing trust assumptions across processes. The array of these fail-prone systems, denoted by $\mathbb{F} = [\mathcal{F}_1, \mathcal{F}_2, \ldots, \mathcal{F}_n]$, is called the *asymmetric fail-prone system*, and it encapsulates the system-wide asymmetric trust relationships within $\mathcal{P}$.

The properties required from a corresponding quorum system for $\mathbb{F}$ are defined below. Observe that, for any pair of processes with correctly specified trust assumptions, meaning that in every execution the actual faulty processes are contained in their respective fail-prone sets, the asymmetric Byzantine quorum system provides the same consistency and availability guarantees as in the symmetric setting, among the quorums of these processes.

Before proceeding, we introduce the following notation, which will be useful in the formal definitions to come. For a given set $\mathcal{A} \subseteq 2^{\mathcal{P}}$ of subsets of $\mathcal{P}$ the set $\mathcal{A}^*$ is defined as $\mathcal{A}^* = \{a \subseteq A \big| A \in \mathcal{A}\}$, the set of all subsets of the sets in $\mathcal{A}$.

**Definition 5 (Asymmetric Byzantine Quorum System).** An *asymmetric Byzantine quorum system* for an asymmetric fail-prone system $\mathbb{F}$ is an array of collections of sets $\mathbb{Q} = [\mathcal{Q}_1, \ldots, \mathcal{Q}_n]$, where $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ for each $i \in [1, n]$. The set $\mathcal{Q}_i$ is called the *quorum system* of process $p_i$, and any set $Q_i \in \mathcal{Q}_i$ is called a *quorum* for $p_i$.

The system satisfies the following properties:

- **Consistency.** The intersection of any two quorums from any two processes contains at least one process that one of them trusts. Formally,

$$\forall i, j \in [1, n], \forall Q_i \in \mathcal{Q}_i, \forall Q_j \in \mathcal{Q}_j, \forall F_{ij} \in \mathcal{F}_i^* \cap \mathcal{F}_j^* : \quad Q_i \cap Q_j \nsubseteq F_{ij}.$$

- **Availability.** For every process $p_i$, and every fail-prone set $F \in \mathcal{F}_i$, there exists a quorum $Q_i \in \mathcal{Q}_i$ disjoint from $F$, i.e.,

$$\forall i \in [1, n], \forall F \in \mathcal{F}_i, \exists Q_i \in \mathcal{Q}_i : Q_i \cap F = \emptyset.$$

The $B^3$-condition is the asymmetric counterpart of the $Q^3$-condition, and it expresses precisely how many faults an asymmetric quorum system can tolerate. The modifications made to its definition reflect the corresponding changes in the consistency property of the quorum system.

**Definition 6 ($B^3$-Condition).** An asymmetric fail-prone system $\mathbb{F}$ satisfies the *$B^3$-condition*, abbreviated as $B^3(\mathbb{F})$, whenever it holds that

$$\forall i, j \in [1, n], \forall F_i \in \mathcal{F}_i, \forall F_j \in \mathcal{F}_j, \forall F_{ij} \in \mathcal{F}_i^* \cap \mathcal{F}_j^* : F_i \cup F_j \cup F_{ij} \neq \mathcal{P}.$$

**Lemma 2 (Alpos *et al.* [1]).** *An asymmetric fail-prone system $\mathbb{F}$ satisfies $B^3(\mathbb{F})$ if and only if there exists an asymmetric quorum system for $\mathbb{F}$.*

For a set of sets $\mathcal{B}$ its complement is denoted by $\bar{\mathcal{B}}$ and is defined as $\bar{\mathcal{B}} = \{\mathcal{P} \setminus B \big| B \in \mathcal{B}\}$.

**Definition 7 (Canonical Quorum System).** An asymmetric quorum system $(\mathbb{F}, \mathbb{Q})$ is called *canonical* if, for every process $p_i$, the quorum system for $p_i$ is the complement of the fail-prone system for $p_i$, i.e.,

$$\forall i \in [1, n] : \mathcal{Q}_i = \bar{\mathcal{F}}_i.$$

The following lemma captures a key intersection property that holds in canonical asymmetric quorum systems but does not generally hold in arbitrary quorum systems.

**Lemma 3.** *For a canonical asymmetric quorum system $(\mathbb{F}, \mathbb{Q})$, the following intersection is non-empty:*

$$\forall i, j \in [1, n],\ \forall Q_{i_1}, Q_{i_2} \in \mathcal{Q}_i,\ \forall Q_{j_1}, Q_{j_2} \in \mathcal{Q}_j : Q_{i_1} \cap Q_{j_1} \cap (Q_{i_2} \cup Q_{j_2}) \neq \emptyset.$$

*Proof.* Since the quorum system is canonical, there exist fault sets $F_i = \mathcal{P} \setminus Q_{i_2} \in \mathcal{F}_i$ and $F_j = \mathcal{P} \setminus Q_{j_2} \in \mathcal{F}_j$. By the consistency property, we have that

$$F_i \cap F_j = (\mathcal{P} \setminus Q_{i_2}) \cap (\mathcal{P} \setminus Q_{j_2}) \subseteq Q_{i_1} \cap Q_{j_1}.$$

This is equivalent to:

$$\mathcal{P} \setminus (Q_{i_2} \cup Q_{j_2}) \subseteq Q_{i_1} \cap Q_{j_1}.$$

Taking complements on both sides, we conclude that

$$Q_{i_1} \cap Q_{j_1} \cap (Q_{i_2} \cup Q_{j_2}) \neq \emptyset,$$

which proves the lemma. □

**Asymmetric Core Sets and Kernels.** The notions of core sets and kernels extend naturally to the asymmetric trust setting. For an asymmetric fail-prone system $\mathbb{F}$, we define the asymmetric core set system $\mathbb{C} = [\mathcal{C}_1, \cdots, \mathcal{C}_n]$, where each $\mathcal{C}_i$ is the core set system corresponding to the local fail-prone system $\mathcal{F}_i$ of process $p_i$.

Similarly, for an asymmetric quorum system $\mathbb{Q}$ corresponding to $\mathbb{F}$, the asymmetric kernel system is denoted by $\mathbb{K} = [\mathcal{K}_1, \cdots, \mathcal{K}_n]$, where each $\mathcal{K}_i$ is the kernel system derived from the local quorum system $\mathcal{Q}_i$ for the processes $p_i \in \mathcal{P}$.

**Definition 8 (Kernel for all Processes).** A *kernel for all processes* $K \subseteq \mathcal{P}$ for an asymmetric kernel system $\mathbb{K}$ is a set of processes such that every process $p_i \in \mathcal{P}$ has a kernel contained in $K$. Formally,

$$\forall p_i \in \mathcal{P},\ \exists K' \in \mathcal{K}_i : K' \subseteq K.$$

**Naive and Wise Processes.** Based on the formalism introduced above, for a given protocol execution and a set $F$ of faulty processes, the correct processes $\mathcal{P} \setminus F$ can be classified into two groups. A process $p_i$ is called *wise* if it correctly anticipates the fault set, that is, if $F \in \mathcal{F}_i^*$. Otherwise, it is called *naive*, meaning $F \notin \mathcal{F}_i^*$.

In the symmetric setting, this distinction does not arise: for any given failure set $F$, either all correct processes are wise or all are naive, depending on whether $F \in \mathcal{F}^*$. This is a key differentiation between the symmetric and asymmetric settings. In symmetric quorum systems, wise processes are always guaranteed by availability to have access to a quorum for themselves consisting only of wise processes. In the asymmetric setting, this is not the case: availability only guarantees the existence of a quorum of correct processes, which may include naive processes.

However, it turns out that in many protocols, naive processes can be easily exploited by the adversary, making it impossible to provide certain security and liveness guarantees for them. This effect snowballs, as wise processes may rely on naive processes in their quorums, thereby weakening the security and liveness guarantees that wise processes can uphold. Recall that in the symmetric case, every wise process has access to a quorum composed entirely of other wise processes. This guarantee is generalized in the asymmetric model through the concept of a *guild*: a set of processes that are wise and contain a quorum for themselves within the guild. Observe that the structure of the guilds is determined by the structure of the quorum system. The fail-prone sets, on the other hand, determine which processes are wise in a given execution and therefore, in a sense, determine which guilds are active under that particular execution.

**Definition 9 (Guild).** Given an asymmetric fail-prone system $\mathbb{F}$, an asymmetric quorum system $\mathbb{Q}$ for $\mathbb{F}$, and a protocol execution with faulty processes $F$, a set of processes $\mathcal{G}$ is called a *guild* for $F$ and $\mathbb{Q}$ if it satisfies the following two properties:

- **Wisdom.** $\mathcal{G}$ consists only of wise processes. Formally,

$$\forall p_i \in \mathcal{G} : F \in \mathcal{F}_i^*.$$

- **Closure.** $\mathcal{G}$ contains a quorum for each of its members, i.e.,

$$\forall p_i \in \mathcal{G}, \exists Q_i \in \mathcal{Q}_i : Q_i \subseteq \mathcal{G}.$$

**Lemma 4 (Alpos *et al.* [1]).** *In any execution with a guild under faults $F$, any two guilds $\mathcal{G}_1$ and $\mathcal{G}_2$ intersect.*

The set of processes that contains all guilds of a given execution is itself a guild and is referred to as the *maximal guild*, denoted by $\mathcal{G}_{\text{max}}$. A process that belongs to the maximal guild is called a *guild member*. For a given guild and one of its members, the term *guild quorum* refers to a quorum (from the member's local quorum system) that lies entirely within the guild.

The following definition introduces the notion of a minimal guild, which is used to characterize the tolerated system, a key concept introduced at the end of this section.

**Definition 10 (Minimal Guild).** Let $(\mathbb{F}, \mathbb{Q})$ be an asymmetric quorum system, and let $F$ be a set of faulty processes. A set $\mathcal{G} \subseteq \mathcal{P}$ is called a *minimal guild* (with respect to $F$) if:

- $\mathcal{G}$ is a guild in executions with faulty processes $F$.
- No strict subset of $\mathcal{G}$ is a guild under faults $F$, i.e.,

$$\forall \mathcal{G}' \subset \mathcal{G},\ \exists p_i \in \mathcal{G}' : \forall Q \in \mathcal{Q}_i,\ Q \not\subseteq \mathcal{G}'.$$

We denote by $\mathbb{G}_F$ the set of all guilds, and by $\mathbb{G}_F^{\text{min}}$ the set of all minimal guilds, in an execution with faults $F \subseteq \mathcal{P}$.

The following lemma establishes that $\mathbb{G}_\emptyset$ contains all the guilds of a quorum system, providing a convenient way to denote the set of all guilds.

**Lemma 5.** *The set of all guilds under all possible executions is equal to the set of guilds under the execution with no faulty processes. Formally,*

$$\bigcup_{F \in 2^{\mathcal{P}}} \mathbb{G}_F = \mathbb{G}_\emptyset.$$

*Proof.* Clearly, $\mathbb{G}_\emptyset \subseteq \bigcup_{F \in 2^{\mathcal{P}}} \mathbb{G}_F$, since $\emptyset \in 2^{\mathcal{P}}$ and thus $\mathbb{G}_\emptyset$ is one of the sets in the union.

Let $F \in 2^{\mathcal{P}}$ be an arbitrary set of faulty processes, and let $\mathcal{G} \in \mathbb{G}_F$ be any guild in an execution with faulty processes $F$. Then $\mathcal{G} \in \mathbb{G}_\emptyset$ as well, since:

- every process in $\mathcal{G}$ is wise when no process is faulty, and
- the closure property still holds, as it is independent of which processes are faulty.

Hence, $\mathbb{G}_F \subseteq \mathbb{G}_\emptyset$. Since $F$ was chosen arbitrarily, it follows that

$$\bigcup_{F \in 2^{\mathcal{P}}} \mathbb{G}_F \subseteq \mathbb{G}_\emptyset.$$

Combining both directions, we conclude:

$$\mathbb{G}_\emptyset = \bigcup_{F \in 2^{\mathcal{P}}} \mathbb{G}_F.$$

□

The following lemmas capture four key properties of kernels and guilds which are used extensively throughout this work.

**Lemma 6 (Alpos *et al.* [1]).** *For a wise process $p_i$ and any of its quorums $Q \in \mathcal{Q}_i$, $Q$ contains a kernel of correct processes for any other wise process $p_j$.*

**Lemma 7.** *In every execution with a guild, for every guild member $p_i \in G_{max}$, each kernel $K_i \in \mathcal{K}_i$ for $p_i$ contains a guild member $p_j \in \mathcal{G}_{max}$. Formally,*

$$\forall p_i \in \mathcal{G}_{max},\ K_i \in \mathcal{K}_i\ \exists p_j \in \mathcal{G}_{max} : p_j \in K_i.$$

*Proof.* Since $K_i$ is a kernel for $p_i$ it intersects with every quorum for $p_i$, hence in particular with its guild quorum $Q_\mathcal{G}$. Therefore $Q_\mathcal{G} \cap K_i \neq \emptyset$, which implies that some guild member $p_j \in \mathcal{G}_{\text{max}}$ is contained in $K_i$. □

**Lemma 8 (Alpos *et al.* [1]).** *In every execution with a guild, the maximal guild $\mathcal{G}_{max}$ is a kernel for all processes.*

**Lemma 9 (Alpos *et al.* [1]).** *In every execution with a guild, every quorum for every correct process contains a guild member.*

**Tolerated System.** Given an asymmetric Byzantine quorum system $(\mathbb{F}, \mathbb{Q})$, the *tolerated system* $\mathcal{T}$ is defined as the complement of all minimal guilds of the quorum system. Since many protocols based on asymmetric quorum systems rely on guilds for their safety and liveness guarantees, the tolerated system characterizes the set of failures the system can endure while still upholding the guarantees of the protocols it runs.

**Definition 11 (Tolerated System).** Let $\mathbb{Q}$ be an asymmetric quorum system. The *tolerated system* $\mathcal{T}$ is

$$\mathcal{T} = \{\mathcal{P} \setminus \mathcal{G} \mid \mathcal{G} \in \mathbb{G}_{\emptyset}^{\min}\}.$$

If the asymmetric quorum system is canonical then the canonical system $(\mathcal{T}, \mathcal{H} = \mathbb{G}_{\emptyset}^{\min})$ forms a symmetric byzantine quorum system.

**Lemma 10 (Alpos *et al.* [1]).** *For a given canonical asymmetric Byzantine quorum system* $(\mathbb{F}, \mathbb{Q})$*, its tolerated system* $(\mathcal{T}, \mathcal{H} = \mathbb{G}_{\emptyset}^{min})$ *is a symmetric quorum system.*

# 4

# Tolerated System: Intractable Construction and Efficient Recognition

In this chapter, we study the computational complexity of the tolerated quorum system $\mathcal{H}$. We show that explicitly enumerating it is intractable in general. We then discuss how the quorum system $\mathcal{H}$ can nevertheless be recognized efficiently. To this end, we introduce the concept of superset recognizers and explain how, given efficient recognizers for the quorum systems $\mathcal{Q}_i$ of the underlying asymmetric quorum system $\mathbb{Q}$ , the implementation of consensus algorithms relying on these systems can be made more practical.

## 4.1 Hardness of Tolerated System Construction

In this section, we show that totally enumerating the tolerated quorum system is in general intractable. To do so, we first introduce the Minimal Set Cover Search Problem, an NP-hard problem which we will reduce to the problem of finding the smallest guild in an execution with no faults. This we then use to establish the desired result.

**Definition 12 (Minimal Set Cover Search Problem).** Given a universe $\mathcal{U} = \{u_1, \ldots, u_n\}$ and a collection of subsets of $\mathcal{U}$, denoted $\mathcal{S} = \{S_1, \ldots, S_m\} \subseteq 2^{\mathcal{U}}$, such that $\bigcup_{S \in \mathcal{S}} S = \mathcal{U}$, find a sub-collection $C \subseteq \mathcal{S}$ such that:

- $C$ **covers** $\mathcal{U}$**.** $\bigcup_{S \in C} S = \mathcal{U}$.
- $C$ **is of minimal size.** For all $C' \subseteq \mathcal{S}$ such that $\bigcup_{S \in C'} S = \mathcal{U}$, we have $|C'| \geq |C|$.

The following definition introduces the Smallest Guild Search Problem. Recall that, by Lemma 5, the set $\mathbb{G}_\emptyset$ denotes all guilds of the quorum system.

**Definition 13 (Smallest Guild Search Problem).** Given an asymmetric quorum system $(\mathbb{F}, \mathbb{Q})$, a set of processes $\mathcal{G} \subseteq \mathcal{P}$ is a solution to the smallest guild search problem if $\mathcal{G} \in \mathbb{G}_\emptyset$ and $\mathcal{G}$ has the least number of processes in $\mathbb{G}_\emptyset$.

**Lemma 11.** *The Smallest Guild Search Problem is NP-hard.*

*Proof.* We prove this lemma by giving a polynomial-time reduction from the Minimal Set Cover Problem to the Smallest Guild Search Problem.

Given an instance $(\mathcal{U}, \mathcal{S})$ of the Minimal Set Cover Problem, we construct an asymmetric quorum system $(\mathbb{F}, \mathbb{Q})$ as follows:

Let $\mathcal{P} = \{p_1, \ldots, p_n, p_{n+1}, \ldots, p_{n+m}\}$. The processes $p_1, \ldots, p_n$ are in one-to-one correspondence with the elements of the universe $u_1, \ldots, u_n$, where process $p_i$ represents element $u_i$ of $\mathcal{U}$ for $1 \leq i \leq n$. The processes $p_{n+1}, \ldots, p_{n+m}$ correspond one-to-one with the sets $S_1, \ldots, S_m$, with process $p_{n+j}$ representing set $S_j$ for $1 \leq j \leq m$.

The fail-prone system is defined as $\mathbb{F} = [\emptyset, \ldots, \emptyset]$, meaning every process anticipates no failures. This simplifies the construction, as the fail-prone sets do not influence the quorum properties. For each process $p_i$ where $1 \leq i \leq n$, its quorum system is defined as

$$\mathcal{Q}_i = \{\{p_1, \cdots, p_n\} \cup \{p_{n+j}\} \big| u_i \in S_j\}.$$

In words, process $p_i$ has a quorum for each set $S_j$ that contains the element $u_i$. Each such quorum includes all processes representing elements in $\mathcal{U}$ and one additional process representing a set in $\mathcal{S}$. For each process $p_{n+j}$ where $1 \leq j \leq m$, its quorum system is:

$$\mathcal{Q}_{n+j} = \{\{p_1, \cdots, p_n\}\}.$$

Clearly, $(\mathbb{F}, \mathbb{Q})$ is an asymmetric Byzantine quorum System. The availability property is satisfied trivially since every process has at least one process. Moreover, every quorum $Q \in \mathcal{Q}_k$ for $p_k$ contains the set $p_1, \ldots, p_n$. Thus, any two quorums intersect, satisfying the consistency condition.

Next we show that this construction is a one-to-one mapping of a set covers of size $l$ onto guilds of size $l + n$. Suppose $\mathcal{G} \in \mathbb{G}_\emptyset$ is a guild of size $n + l$. Since every quorum contains $p_1, \ldots, p_n$, it follows that $p_1, \ldots, p_n \subseteq \mathcal{G}$. Let $C_p = \mathcal{G} \setminus \{p_1, \ldots, p_n\}$ be the subset of processes representing sets in $\mathcal{S}$, and let $C = \{S_j \mid p_{n+j} \in C_p\}$. Clearly, $|C| = l$.

To show that $C$ is a set cover of $\mathcal{U}$, take any $u_i \in \mathcal{U}$. Then $p_i \in \mathcal{G}$, so by the closure property, there exists a quorum $Q \in \mathcal{Q}_i$ for $p_i$ such that $Q \subseteq \mathcal{G}$. Each such $Q$ includes some $p_{n+j}$ with $u_i \in S_j$, hence $S_j \in C$. Since $i$ was arbitrary, $C$ covers all of $\mathcal{U}$.

Conversely, let $C$ be a set cover of size $l$. Define: $\mathcal{G}_C = \{p_1, \cdots, p_n\} \cup \{p_{n+j} \big| S_j \in C\}$. To show $\mathcal{G}_C \in \mathbb{G}_\emptyset$, observe:

For each $p_{n+j} \in \mathcal{G}_C$, its quorum $\{p_1, \ldots, p_n\}$ is contained in $\mathcal{G}_C$. For each $p_i$, since $C$ is a cover, there exists $S_j \in C$ such that $u_i \in S_j$. Then $Q = \{p_1, \ldots, p_n, p_{n+j}\} \in \mathcal{Q}_i$ and $Q \subseteq \mathcal{G}_C$. Thus, $\mathcal{G}_C$ satisfies the closure property and is a guild in the execution with no faulty processes.

Therefore the construction reduces the problem of finding the smallest set cover to that of finding the smallest guild, since every guild of size $n + l$ stands in one-to-one correspondence with a set cover of size $l$, and there fore a guild of minimal size must correspond to a set cover of smallest size. □

A similar result has previously been established for federated Byzantine agreement systems [13], where the authors proved that finding the smallest quorum of an FBAS is NP-hard.

The following lemma concludes this section, establishing that the tolerated quorum system cannot be constructed efficiently.

**Lemma 12.** *Constructing the tolerated quorum system $\mathcal{H}$ cannot be done in polynomial time with respect to the size of the quorum system.*

*Proof.* This follows from the fact that the smallest guild $\mathcal{G}_{\text{min}}$ (with respect to cardinality) is a minimal guild, and thus is contained in the set $\mathbb{G}_\emptyset^{\text{min}} = \mathcal{H}$.

Suppose that $\mathcal{H}$ could be constructed in polynomial time in the size of the quorum system (i.e., there exists an algorithm with polynomial total time, and the number of minimal guilds is polynomially bounded). Then we could solve the smallest guild search problem by iterating over all elements of $\mathcal{H}$ and selecting the one with minimal size, which would also run in polynomial time.

However, the smallest guild search problem is NP-hard, as shown above by a reduction from Set Cover. Hence, unless P = NP, there can be no such polynomial-time construction algorithm for $\mathcal{H}$.

Therefore, constructing $(\mathcal{T}, \mathcal{H})$ cannot be done efficiently. □

Since explicitly enumerating the tolerated quorum system $(\mathcal{T}, \mathcal{H})$ is, in general, inefficient, it is impractical to base algorithms on such enumeration.

However, as the following lemmas in the next section demonstrate, it is possible to efficiently recognize sets in $(\mathcal{T}, \mathcal{H})$ by leveraging efficient superset recognizers for the individual quorum systems $\mathcal{Q}_i$ of the asymmetric quorum system $\mathbb{Q}$.

## 4.2 Superset Recognizers

Quorum-based protocols are usually described as event-driven protocols, where the quorum system is used to determine whether messages satisfying a given condition have been received from a quorum of processes (examples can be seen in Algorithm 2 line 18). The same principle applies to kernel and coreset systems. To operationalize these conditions, we introduce the abstraction of a superset recognizer: an algorithm that recognizes supersets of a given collection of sets (as defined in Definition 14).

There is no general recipe for implementing a waiting condition using superset recognizers; however, in all cases we have encountered, this abstraction has proven sufficient.

**Definition 14 (Superset Recognizer).** For a given set of sets of processes $\mathcal{A}$, a *superset recognizer* $R_{\mathcal{A}}$ is an algorithm with binary output, that detects whether a set $X$ contains a subset in $\mathcal{A}$. Formally,

$$\forall X \subseteq \mathcal{P} : R_{\mathcal{A}}(X) = \text{TRUE} \iff \exists A \subseteq X : A \in \mathcal{A}.$$

We show that a superset recognizer for the quorum systems $\mathcal{Q}_i$ of an asymmetric quorum system can be efficiently transformed into recognizers for its kernel systems $\mathcal{K}_i$ and for the quorum system $\mathcal{H}$ of its tolerated system. Consequently, the presented algorithms can be implemented entirely on top of efficient superset recognizers for the individual quorum systems $\mathcal{Q}_i$.

**Lemma 13.** *Given a superset recognizer $R_{\mathcal{Q}}$ for a quorum system $\mathcal{Q}$, the kernel system $\mathcal{K}$ of $\mathcal{Q}$ is recognized by the superset recognizer $R_{\mathcal{K}}$, which for any set $X \subseteq \mathcal{P}$, returns $\neg R_{\mathcal{Q}}(\mathcal{P} \setminus X)$.*

*Proof.* Given a set $X$, if $R_{\mathcal{K}}(X) = \text{FALSE}$, then by definition $R_{\mathcal{Q}}(\mathcal{P} \setminus X) = \text{TRUE}$. Therefore, there exists a quorum $Q \in \mathcal{Q}$ such that $Q \subseteq \mathcal{P} \setminus X$, which implies that $Q \cap X = \emptyset$. Hence $X$ cannot contain a kernel.

If $R_{\mathcal{K}}(X) = \text{TRUE}$, then $R_{\mathcal{Q}}(\mathcal{P} \setminus X) = \text{FALSE}$. Let $Q \in \mathcal{Q}$. Since $R_{\mathcal{Q}}(\mathcal{P} \setminus X) = \text{FALSE}$, it follows that $Q \not\subseteq \mathcal{P} \setminus X$, which implies that $X \cap Q \neq \emptyset$.

This establishes that $R_{\mathcal{K}}$ is a superset recognizer for $\mathcal{K}$. □

**Algorithm 1** Superset Recognizer for the set of Guilds $\mathbb{G}_\emptyset$

**Input:**
1: $X$ // Set of processes
2: $[R_{\mathcal{Q}_1}, \cdots, R_{\mathcal{Q}_n}]$ // Array of superset recognizers for the quorum system of interest

3: **function** $R_{\mathcal{H}}(X, [R_{\mathcal{Q}_1}, \cdots, R_{\mathcal{Q}_n}])$ **returns** Boolean **is**
4: **while** $\exists p_i \in X : \neg R_{\mathcal{Q}_i}(X)$ **do**
5: $X \leftarrow X \setminus \{p_i\}$
6: **return** $X \neq \emptyset$

**Lemma 14.** *Given an asymmetric quorum system $\mathbb{Q}$ and superset recognizers $R_{\mathcal{Q}_i}$ for its individual quorum systems $\mathcal{Q}_i$, Algorithm 1 implements a superset recognizer for $\mathbb{G}_\emptyset$, and hence also for $\mathbb{G}_\emptyset^{min}$.*

*Proof.* Let $X \in \mathcal{P}$. If $X$ contains a guild, then there exists a set $G \subseteq X$ such that for all $p_i \in G$, it holds that $R_{\mathcal{Q}_i}(G) = \text{TRUE}$. Therefore, the while loop of $R_{\mathcal{H}}$ never removes any element in $G$, and hence $R_{\mathcal{H}}(X) = \text{TRUE}$.

If $X$ does not contain a guild, then for all $X' \subseteq X$, there exists a process $p_i \in X'$ such that $R_{\mathcal{Q}_i}(X') = \text{FALSE}$. Hence, the while loop removes all elements from $X$, and $R_{\mathcal{H}}(X) = \text{FALSE}$.

Since one element is removed from $X$ n each iteration of the loop, and the empty set does not satisfy the loop condition, the algorithm terminates after at most $|X|$ iterations. □

# 5 Asynchronous Consensus

In this chapter, we present a consensus algorithm for the fully asynchronous communication model. We begin by defining the Asymmetric Strong Byzantine Consensus abstraction that the algorithm implements. We then introduce the two core building blocks it relies on: the Asymmetric Common Coin and the Asymmetric Binary Validated Broadcast (ABV-broadcast). Finally, we describe the algorithm proposed by Alpos *et al.* [1], a generalization of the asynchronous protocol by Mostéfaoui, Moumen, and Raynal [19] to the asymmetric trust setting. For clarity, we refer to this algorithm as Asymmetric Randomized Consensus throughout the remainder of this thesis.

## 5.1 Asymmetric Strong Byzantine Consensus Specification

Consensus is a fundamental problem in distributed computing, in which a group of processes must agree on a single value even if they start with different proposals and some processes fail or behave maliciously. At an abstract level, each process interacts with the consensus service by submitting a value through a *Propose*$(v)$ event and, at some later point, receiving a *Decide*$(v)$ event indicating the agreed outcome. This interaction hides the complexity of coordination, fault tolerance, and message exchange, providing the processes with a simple and consistent interface for reaching agreement.

Consensus abstractions come in different flavours, offering varying strengths of validity, agreement, and termination guarantees depending on the fault model and system assumptions. The Asymmetric Randomized Consensus algorithm described here implements one such variant: the Asymmetric Strong Byzantine Consensus interface specified in Module 1. The *Agreement* property captures the core requirement of consensus, while *Strong Validity* ensures that the decided value was indeed proposed by a correct process, more precisely, by a guild member, if such a guild exists. In a later section, we relax this to *Weak Validity*, which permits the decision of arbitrary values in executions where at least one faulty process is present.

The *Probabilistic Termination* condition is essential for consensus in a fully asynchronous setting. As shown by Fischer, Lynch, and Paterson [10], no deterministic algorithm can guarantee termination under asynchrony and even a single crash fault. Instead, randomized algorithms ensure that, for any finite round number $r$, the probability of non-termination after round $r$ can be made arbitrarily small, and, in the limit, goes to zero.

## 5.2 Asymmetric Common Coin Abstraction

The required non-determinism of the algorithm is encapsulated in a common coin abstraction, whose specification is shown in Module 2. This abstraction provides processes with access to a random coin shared among them. When a process wants to obtain the coin's value, it triggers the *ReleaseCoin* event. Once a sufficient number of processes have released the coin, the process is informed of the coin's value via the *CoinOutput* event.

The *Unpredictability* and *No Bias* properties of the coin ensure that the adversary cannot deduce the coin's value before it is too late. The *Matching* property guarantees that all processes observe the same random value.

In this work, no implementation of the Asymmetric Common Coin abstraction is provided, as the algorithm proposed by Alpos et al. [1] is deemed impractical.

**Module 1** Interface and Properties of the Asymmetric Strong Byzantine Consensus abstraction

**Module:**

Name: AsymmetricStrongByzantineConsensus, **instance** *asbc*.

**Events:**

**Request**: *asbc.Propose*$(v)$: Propose value $v$.
**Indication**: *asbc.Decide*$(v)$: Decision value of the protocol.

**Properties:**

**ASBC1:** *Probabilistic Termination*: In all executions with a guild, every wise process decides with probability 1, meaning that $\lim_{r\to\infty} P\big(\text{wise process } p_i \text{ decides by round } r\big) = 1$.

**ASBC2:** *Strong Validity*: In every execution with a guild, a wise process only decides a value that has been proposed by a process in the maximal guild.

**ASBC3:** *Integrity*: No correct process decides more than once.

**ASBC4:** *Agreement*: No two wise processes decide differently.

**Module 2** Interface and Properties of the Asymmetric Common Coin Abstraction

**Module:**

Name: AsymmetricCommonCoin, **instance** *acc*.

**Events:**

**Request**: *acc.ReleaseCoin*$()$: Signals that the process is ready to receive the coin's value.
**Indication**: *acc.OutputCoin*$(c)$: Delivers the coin value $c$ to the process.

**Properties:**

**ACC1:** *Termination*: In all executions with a guild, every process in the maximal guild eventually outputs a coin value.

**ACC2:** *Unpredictability*: In every execution with a guild, no process has information about the coin's value before at least one kernel for every process of all correct processes has released the coin.

**ACC3:** *Matching*: In every execution with a guild, all processes in the maximal guild output the same coin value.

**ACC4:** *No Bias*: The output of the coin is uniformly distributed.

## 5.3 Asymmetric Binary Validated Broadcast

The communication in the consensus algorithm is carried out using a broadcast primitive called Asymmetric Binary Validated Broadcast. Its interface and properties are defined in Module 3. Each process may broadcast a value $b \in \{0, 1\}$ by invoking *abv.Broadcast*$(b)$.
The *Termination* property ensures that every wise process eventually delivers some value. However, the primitive may trigger two deliver events, one for each binary value. The *Validity* condition ensures that if a sufficient number of processes broadcast the same value, then that value is eventually delivered by every wise process. The *Integrity* property guarantees that any delivered value was broadcast by at least one member of the maximal guild. Finally, the *Agreement* property ensures that all wise processes eventually deliver the same set of values.

**Module 3** Interface and Properties of the Asymmetric Binary Validated Broadcast Abstraction

**Module:**

Name: AsymmetricBinaryValidatedBroadcast, **instance** *abv*.

**Events:**

**Request**: *abv.Broadcast*$(b)$: Broadcast bit $b$ to peers.
**Indication**: *abv.Delive*$(b)$: Deliver bit $b$, broadcast by some peers.

**Properties:**

**ABV1:** *validity*: In all executions with a guild, let $K$ be a kernel for all processes, consisting of correct processes that has broadcast the same value $b \in \{0, 1\}$. Then, every wise process eventually delivers $b$.

**ABV2:** *Integrity*: In every execution with a guild, if a wise process delivers some $b$, then $b$ has been broadcast by some process in the maximal guild.

**ABV3:** *Agreement*: In every execution with a guild, if a wise process delivers some value $b$, then every wise process eventually delivers $b$.

**ABV4:** *Termination*: In every execution with a guild, every wise process eventually delivers some value.

The implementation of the Asymmetric Binary Validated Broadcast is shown in Algorithm 2. When a *abv.Broadcast*$(b)$ event is triggered, the process sends the value $b$ to all its peers in a $[\text{VALUE}, b]$ message. Every process maintains a local data structure *values*, in which it records, for each of its peers, the values $b \in \{0, 1\}$ for which it has received a $[\text{VALUE}, b]$ message. Once the process receives the same value $b$ from a kernel for itself, it joins the broadcast by also sending a $[\text{VALUE}, b]$ message to its peers. When it has received the value $b$ from a quorum for itself, it delivers the value. This mechanism of amplifying a broadcast upon receiving matching messages from a kernel for itself, and deciding or delivering upon receiving matching messages from a quorum for itself, is a common design pattern that recurs in multiple algorithms throughout this thesis.

## 5.4 Asymmetric Randomized Consensus Algorithm

The Asymmetric Randomized Consensus algorithm is shown in Algorithm 3 [1]. As mentioned, the algorithm proceeds in rounds. Each round runs an independent instance of Asymmetric Binary Validated Broadcast, identified by a tag corresponding to the current round. When a value is proposed, it is broadcast, and the process waits for a deliver event to occur.

Once the process delivers a value $b$, it records the values it has delivered in the current round and sends an $[\text{AUX}, b]$ message to all its peers. When the process has received a set $B \subseteq \{0, 1\}$ of values carried by AUX messages from a quorum for itself, it releases the coin. The process then waits for the coin value $s$. Note that the set $B$ may change while the process waits for the coin to return a value. Once this occurs, the process checks whether there is a single value $b$ in $B$. If so, and if $b$ matches the coin's output $s$, the process becomes ready to decide $b$, and does so by broadcasting a decide message $[\text{DECIDE}, b]$ to all its peers. It then proceeds to the next round with $b$ as its decision value. If $B$ contains both values, the process adopts $s$, the coin value, as its decision value for the next round.

In parallel, the protocol may disseminate DECIDE messages and terminate. When the process receives matching DECIDE messages for some value $b$ from a kernel for itself, it amplifies the decision by sending a DECIDE message of its own for $b$. Once it receives DECIDE messages for $b$ from a quorum for itself, it triggers a decide event and terminates.

**Algorithm 2** Asymmetric Binary Validated Broadcast Implementation (Process $p_i$).

**implements**
AsymmetricBinaryValidatedBroadcast, **instance** *abv*
**state**
7: *sentvalue* $\leftarrow [\text{FALSE}]^2$ // Entry *sentvalue*$[b]$ indicates whether $p_i$ has sent $[\text{VALUE}, b]$
8: *values* $\leftarrow [\emptyset]^n$ // List of sets of received binary values

9: **upon event** *abv.Broadcast*$(b)$ **do**
10: *sentvalue*$[b] \leftarrow$ TRUE
11: sned message $[\text{VALUE}, b]$ to all $p_j \in \mathcal{P}$

12: **upon** receiving message $[\text{VALUE}, b]$ from $p_j$ **do**
13: **if** $b \notin$ *values*$[j]$ **then**
14: *values*$[j] \leftarrow$ *values*$[j] \cup \{b\}$

15: **upon** $\exists b \in \{0, 1\}$ **such that** $\{p_j \in \mathcal{P} \big| b \in \textit{values}[j]\} \in \mathcal{K}_i \wedge \neg \textit{sentvalue}[b]$ **do**
16: *sentvalue*$[b] \leftarrow$ TRUE
17: send message $[\text{VALUE}, b]$ to all $p_j \in \mathcal{P}$

18: **upon** $\exists b \in \{0, 1\}$ **such that** $\{p_j \in \mathcal{P} \big| b \in \textit{values}[j]\} \in \mathcal{Q}_i$ **do**
19: **trigger** *abv.Deliver*$(b)$

**Algorithm 3** Asymmetric Randomized Consensus Implementation (Process $p_i$).

**implements**
AsymmetricStrongByzantineConsensus, **instance** *asbc*
**state**
20: *round* $\leftarrow 0$
21: *values* $\leftarrow \{\}$ // Set of delivered binary values for the round
22: *aux* $\leftarrow [\{\}]^n$ // Stores sets of values received in AUX messages in the round
23: *decided* $\leftarrow [\perp]^n$ // Stores binary values that have been reported as decided by other processes
24: *sentdecide* $\leftarrow$ FALSE // Indicates whether $p_i$ has sent a DECIDE message

25: **upon event** *asbc.Propose*$(b)$ **do**
26: **trigger** *abv.Broadcast*$(b)$ with tag *round*

27: **upon event** *abv.Deliver*$(b)$ with tag $r$ **such that** $r =$ *round* **do**
28: *values* $\leftarrow$ *values* $\cup \{b\}$
29: send message $[\text{AUX}, \textit{round}, b]$ to all $p_j \in \mathcal{P}$

30: **upon** receiving a message $[\text{AUX}, r, b]$ from $p_j$ **such that** $r =$ *round* **do**
31: *aux*$[j] \leftarrow$ *aux*$[j] \cup \{b\}$

32: **upon** $\exists \{p_j \in \mathcal{P} \big| \textit{aux}[j] \subseteq \textit{values}\} \in \mathcal{Q}_i$ **do**
33: **trigger** *ReleaseCoin*() with tag *round*

34: **upon event** *acc.OutputCoin*$(s)$ with tag *round*
35: **and** $\exists B \subseteq \{0,1\}, Q_i \in \mathcal{Q}_i$ **such that** $B \neq \emptyset \wedge \forall p_j \in Q_i : B =$ *aux*$[j]$ **do**
36: *round* $\leftarrow$ *round* $+ 1$
37: **if** $\exists b$ **such that** $B = \{b\}$ **then**
38: **if** $b = s \wedge \neg$*sentdecide* **then**
39: send message $[\text{DECIDE}, b]$ to all $p_j \in \mathcal{P}$
40: *sentdecide* $\leftarrow$ TRUE
41: **trigger** *abv.Broadcast(b)* with tag *round*
42: **else**
43: **trigger** *abv.Broadcast*$(s)$ with tag *round*
44: *values* $\leftarrow \{\}$
45: *aux* $\leftarrow [\{\}]^n$

46: **upon** receiving a message $[\text{DECIDE}, b]$ from $p_j$ **such that** *decided*$[j] = \perp$ **do**
47: *decided*$[j] \leftarrow b$

48: **upon** $\exists b \neq \perp$ **such that** $\{p_j \in \mathcal{P} \big| \textit{decided}[j] = b\} \in \mathcal{K}_i$ **do**
49: **if** $\neg$*sentdecide* **then**
50: send message $[\text{DECIDE}, b]$ to all $p_j \in \mathcal{P}$
51: *sentdecide* $\leftarrow$ TRUE

52: **upon** $\exists b \neq \perp$ **such that** $\{p_j \in \mathcal{P} \big| \textit{decided}[j] = b\} \in \mathcal{Q}_i$ **do**
53: *asbc.Decide*$(b)$
54: **halt**

# 6
# Partially Synchronous Consensus

In this chapter, we present an adaptation of the PBFT algorithm, as described by Cachin et al. [6], to the asymmetric trust setting. PBFT itself operates in the partially synchronous model and is a leader-driven consensus algorithm. Leader-driven algorithms are characterized by an unequal distribution of roles among the processes: one process acts as the dedicated leader, which collects all required information from the replicas, decides on a value, and then communicates that decision back to the replicas. The replicas verify the work of the leader, ensuring that the leader's chosen value is valid. This design reduces communication complexity in the decision phase from quadratic (all-to-all) to linear (replicas-to-leader).

However, because the dedicated leader may be faulty, the algorithm must be able to detect and replace it with a correct leader. To achieve this, PBFT is built on two main abstractions:

- **Epoch Change (Heartbeat) Abstraction.** This abstraction, often called the heartbeat mechanism, detects faulty leaders and triggers leader replacement. It produces a sequence of epochs, each with a globally known dedicated leader.
- **Epoch Consensus Abstraction.** This is the core consensus mechanism. It ensures that once a process decides on a value in some epoch, no process can decide on a different value in any later epoch, thereby preserving the *agreement* property of consensus.

APBFT implements a weaker form of consensus, called Asymmetric Weak Consensus (shown in Module 4), which differs from Asymmetric Strong Consensus primarily in its validity property. The weak consensus abstraction provides the same validity guarantee as strong consensus does when all processes are correct, but provides no such guarantee otherwise, allowing decision values that may not have been proposed by any correct process if faulty processes exist. This relaxation is necessary because a faulty leader can behave indistinguishably from a correct leader to its peers, enabling it to cause the processes to decide on a value that was not proposed by any correct process.

The termination conditions differ because the strong consensus algorithm we consider operates in a fully asynchronous model, requiring probabilistic termination to ensure progress. In contrast, APBFT assumes a partially synchronous model, which allows for deterministic termination and stronger guarantees. Thus, the difference in termination stems from the specific system models these algorithms target, rather than the inherent strength of the consensus properties.

In the following sections, we first present the Asymmetric Epoch Change abstraction. When introducing the epoch consensus abstraction, we begin with the symmetric case, and subsequently generalize it to the asymmetric case to aid understanding. In the asymmetric setting, we consider two variations of PBFT: PAPBFT (Partially Asynchronous PBFT), introduced by Zanolini [23], and our own variant, APBFT (Asymmetric PBFT), which is presented in detail in this work.

## 6.1 Asymmetric Epoch Change Primitive

This section presents the Asymmetric Epoch Change abstraction. We first provide its formal specification, and then describe a concrete algorithm based on a round-robin leader selection, along with a discussion of its correctness.

**Module 4** Interface and Properties of Asymmetric Weak Byzantine Consensus

**Module:**
Name: AsymmetricWeakByzantineConsensus, **instance** *awc*.

**Events:**
**Request**: *awc.Propose*($v$): Propose the decision value $v$.
**Indication**: *awc.Decide*($v$): Signals the agreed upon decision value.

**Properties:**
**WC1:** *Termination*: In every execution with a guild, every guild member eventually decides.

**WC2:** *Weak Validity*: If all processes are correct and some process decides $v$, then $v$ was proposed by some process.

**WC3:** *Integrity*: No correct process decides twice.

**WC4:** *Agreement*: No two wise processes decide differently.

### 6.1.1 Asymmetric Epoch Change Specification

The epoch change abstraction signals the beginning of a new epoch by triggering an *StartEpoch*($ts, \ell$) event whenever the current leader is suspected to be faulty. This event carries two parameters: the epoch timestamp *ts* and the designated leader $\ell$ for that epoch.

When a process suspects the leader is faulty, it can trigger the *Complain*(*ts*) event to notify the epoch change abstraction and request a leader change. Once a sufficient number of processes have complained, a new epoch begins.

The timestamps associated with *StartEpoch*($ts, \ell$) must be strictly increasing, and all processes must agree on the same leader for each epoch timestamp: this is enforced by the *Monotonicity* and *Consistency* properties of the Asymmetric Epoch Change interface (see Module 5).

*Putch Resistance* guarantees that no process advances to epoch $ts + 1$ unless at least one guild member has complained about the leader of epoch *ts*, preventing faulty or naive processes from causing unwanted leader changes. *Eventual Progression* ensures that once a kernel for all processes has complained in epoch *ts*, all guild members eventually enter epoch $ts + 1$. *Unbounded Leadership* ensures the existence of an epoch during which the processes remain in the same epoch for a sufficient amount of time to allow termination of the epoch consensus algorithm. Finally, *Uniform Leadership* guarantees that every process is elected leader infinitely often. In combination with *Unbounded Leadership*, this ensures that eventually processes will trust a adequate leader for an arbitrarily long duration.

### 6.1.2 Algorithm Description: Asymmetric Rotating Epoch Change

Algorithm 4 implements the Asymmetric Epoch Change abstraction. It maintains a continuously increasing epoch counter and deterministically derives the leader via a round-robin function *leader*(*ets*) with respect to the process identifiers. When the higher-level protocol triggers *aec.Complain*(*ts*) and *ts* is the current epoch, a process broadcasts a COMPLAINT message. As soon as it receives a quorum for itself of complaints against the current leader, it increments its epoch. Furthermore, if it receives a kernel for itself of complaints but has not yet complained itself in the current epoch, it joins the complainers by sending its own COMPLAINT message. This ensures that (1) Byzantine processes alone cannot force a leader change, and (2) once a guild member advances to the next epoch, all other guild members eventually follow (Lemma 15).

Note that this guarantee applies only to guild members, not to all wise processes: naive processes may become stuck in lower epochs, and if they appear in a wise process's quorum, they can block its progression. This is a critical limitation of the asymmetric epoch change algorithm since, as we will see in the APBFT algorithm, correct processes may hold information essential for a leader of an epoch to make a valid decision.

The algorithm is based on the Rotating Byzantine Leader Detector algorithm presented in the book by Cachin, Guerraoui, and Rodriguez [6]. The interface of the Asymmetric Epoch Change abstraction also follows the Byzantine Epoch Change Module introduced in the same book. In the paper by Bravo et al. [4], the authors point out a flaw in the construction of the epoch change specification [6], where the authors make the assumption (which they justify informally) that every process complains only a finite number of times, stating the properties of the module based

**Module 5** Interface and Properties of the Asymmetric Epoch Change Abstraction

**Module:**

Name: AsymmetricEpochChange, **instance** *aec*.

**Events:**

**Request**: *aec.Complain*$(ts)$: Requests a leader change in epoch *ts*.

**Indication**: *aec.StartEpoch*$(ts, \ell)$: Signals the start of a new epoch.

**Properties:**

**EC1:** *Monotonicity*: If a correct process starts an epoch $(ts, \ell)$ and later starts an epoch $(ts', \ell')$, then $ts' \geq ts$.

**EC2:** *Consistency*: If a correct process starts an epoch $(ts, \ell)$ and another correct process starts an epoch $(ts', \ell')$ with $ts = ts'$, then $\ell = \ell'$.

**EC3:** *Putch Resistance*: A guild member only enters epoch $ts + 1$ if a guild member has previously complained in epoch *ts*.

**EC4:** *Eventual Progression*: If a kernel for all processes complains in epoch *ts*, then all guild members eventually start epoch $ts + 1$.

**EC5:** *Eventual Entry*: In every execution with a guild, if a wise process starts a new epoch *ets*, then eventually every guild member starts epoch *ets*.

**EC6:** *Unbounded Leadership*: In every execution with a guild, if all guild members start an infinite number of epochs, and the time guild members wait before complaining after starting a new epoch is strictly increases with the epoch number, then for every duration $d \in \mathbb{R}$ there exists an epoch *ets* such that all guild members are simultaneously in epoch $ets'$ for at least $d$ time units, for all epochs $ets' \geq ets$.

**EC7:** *Uniform Leadership*: If a guild member starts an infinite number of epochs, then every process is the leader of some epoch infinitely often.

on this assumption. This leads to problems with circular reasoning. To avoid this problem in our description of the epoch change abstraction, the *Eventual Leadership* property is removed and replaced by the weaker *Eventual Entry* property, which holds without assuming that the processes complain only a finite number of times. Furthermore, the *Unbounded Leadership* property is added to capture the essence of the informal argument made by the authors of the book. In this way, it becomes the task of the caller of the epoch change interface to prove that the *Complain* event is triggered only a finite number of times, avoiding circularity.

In the following discussion, a process is said to start an epoch *ets* at some time $t$, when it triggers the event *aec.StartEpoch*$(ets, \cdot)$ at time $t$.

**Algorithm 4** Rotating Byzantine Epoch Change (Process $p_i$).

```
    implements
        AsymmetricEpochChange, instance aec
    state
55:     ets ← 0
56:     complaintset ← [FALSE]^n
57:     complained ← FALSE

58: upon event aec.Init() do
59:     trigger aec.StartEpoch(1, leader(1))

60: upon event aec.Complain(ts) such that ¬complained ∧ ts = ets do
61:     complained ← TRUE
62:     send message [COMPLAINT, ets] to all p_j ∈ P

63: upon receiving a message [COMPLAINT, r] from p_j such that r = ets do
64:     complaintset[j] ← TRUE

65: upon ¬complained ∧ {p_j ∈ P | complaintset[j] = TRUE} ∈ K_i do // Kernel of processes has complained
66:     complained ← TRUE
67:     send message [COMPLAINT, ets] to all p_j ∈ P

68: upon complained ∧ {p_j ∈ P | complaintset[j] = TRUE} ∈ Q_i do // Quorum of processes has complained
69:     ets ← ets + 1
70:     complaintset ← [FALSE]^n
71:     complained ← FALSE
72:     trigger aec.StartEpoch(ets, leader(ets))
```

### 6.1.3 Algorithm Correctness

We now prove that Algorithm 4 implements the Asymmetric Epoch Change interface (Module 5).

**Lemma 15.** *Algorithm 4 satisfies the Eventual Entry property.*

*Proof.* Proof by induction on the epoch *ets*.
**Base Case (*ets* $= 0$):** By initialization, every correct process triggers *aec.startEpoch*$(0, leader(0))$. Therefore, the property holds for *ets* $= 0$.
**Inductive Step:** Assume that for some *ets* $\geq 0$, if any wise process triggers an *aec.startEpoch*$(ets, \ell_{ets})$ event, then eventually every process in $\mathcal{G}_{\text{max}}$ triggers *aec.startEpoch*$(ets, \ell_{ets})$. We now show that if any wise process $p_i$ triggers an *aec.startEpoch*$(ets + 1, \ell)$ event, then eventually every process in $\mathcal{G}_{\text{max}}$ triggers *aec.startEpoch*$(ets + 1, \ell)$. By the protocol, a process can only transition to a new epoch after it has received a quorum $Q$ for itself of complaint messages in epoch *ets*, when handling the event on line 68. Hence, for $p_i$ to start epoch *ets* $+ 1$, it must have previously been in epoch *ets*. By the inductive hypothesis, since $p_i$ (a wise process) has previously triggered *aec.startEpoch*$(ets, \ell_{ets})$, every process in $\mathcal{G}_{\text{max}}$ eventually triggers *aec.startEpoch*$(ets, \ell_{ets})$. Moreover, by Lemma 6 (regarding quorum $Q$), every guild member eventually receives a kernel for itself of complaints for epoch *ets*. Consequently, every process in $\mathcal{G}_{\text{max}}$ eventually sends a complaint message to all processes in epoch *ets* and sets *complained* to TRUE (line 65). Therefore, every process in $\mathcal{G}_{\text{max}}$ eventually receives the necessary complaints (i.e., its quorum condition is satisfied; line 68) in epoch *ets*, causing them to trigger *aec.startEpoch*$(ets + 1, \ell)$ . □

**Lemma 16.** *If a guild member $p_g$ enters epoch $ts > 0$ at time $t$, then for every guild member $p_{g'}$ and every epoch $0 \leq ts' < ts$, there exists a kernel for itself of correct processes $K' \in \mathcal{K}_{g'}$ that has sent a complaint message in epoch $ts'$ to $p_{g'}$ at some time $< t$.*

*Proof.* If $p_g$ starts epoch *ts*, it must have previously started every epoch $0 < ts' \leq ts$ at some time $t' < t$. $p_g$ only starts epoch *ts'* after receiving complaint messages for epoch $ts' - 1$ from a quorum $Q_{ts'}$ for itself, and after it has itself sent a complaint message in epoch $ts' - 1$(line 68).

A correct process only sends a complaint message for epoch $ts' - 1$ to $p_g$ if it also sends one to every process. Therefore, for every guild member $p_{g'}$, by Lemma 6, a kernel for $p_{g'}$ of correct processes has sent a complaint message for epoch $ts' - 1$ to $p_{g'}$ at some time $< t' < t$.

Since *ts'* is chosen arbitrarily in the range $0 < ts' \leq ts$, it follows that for every guild member $p_{g'}$, a kernel for $p_{g'}$ of correct processes has sent complaint messages to $p_{g'}$ in every epoch *ts'* at some time $< t' < t$, with $0 \leq ts' < ts$. □

**Definition 15 (Guild Timestamps).** Let $ETS_t = \{ets \mid ets \in \mathbb{N} \wedge \text{some guild member is in epoch } ets \text{ at time } t\}$ denote the set of epochs of the guild members at time $t$

**Definition 16 (Min/Max Timestamp).** Let $ets_t^{\min} = \min\ ETS_t$ and $ets_t^{\max} = \max\ ETS_t$ denote the minimal and maximal epochs of some guild member at time $t$.

**Lemma 17 (Fast Progression).** *For any time $t$ after GST, if $ets_t^{min} < ets_t^{max}$, then $ets_{t+2\Delta}^{min} \geq ets_t^{min} + 1$.*

*Proof.* Since $t \geq GST$ and by Lemma 16 (which is applicable since $ets_t^{\min} < ets_t^{\max}$), every process in epoch $ets_t^{\min}$ satisfies the kernel condition on line 65 at some time $\leq t + \Delta$, and sends a complaint message, or has already sent a complaint message for that epoch.

Therefore, at time $t' \geq t + \Delta$, every guild member has, in fact, sent a complaint message for epoch $ets_t^{\min}$, since processes can only advance from an epoch after sending a complaint message.

As a result, at time $t^* \leq t + 2\Delta$, every process in epoch $ets_t^{\min}$ satisfies the quorum condition on line 68, forcing them to move on to epoch $ets_t^{\min} + 1$.

Thus, $ets_t^{\min} + 1 \leq ets_{t+2\Delta}^{\min}$. □

**Lemma 18 (Unbounded Leadership).** *Algorithm 4 satisfies the Unbounded Leadership property.*

*Proof.* To prove the lemma, we first have to establish that, at some point in time after *GST*, all guild members catch up to each other and are in the same epoch infinitely often. For this, let $t'$ be the time after *GST* at which every guild member waits at least $7\Delta$ time units after starting an epoch before complaining. Such a time exists because, by assumption, the waiting time before complaining is strictly increasing with every new epoch a guild member starts, and every guild member starts infinitely many epochs. Let $ts_k^{\max} = ets_{t'+k\cdot 7\Delta}^{\max}$ and $ts_k^{\min} = ets_{t'+k\cdot 7\Delta}^{\min}$. Since every guild member waits at least $7\Delta$ time units before complaining, it follows that $ts_{k+1}^{\max} \leq ts_k^{\max} + 1$. Also, by Lemma 17, $ts_{k+1}^{\min} + 2 > ts_k^{\min}$ while $ets_t^{\min} < ets_t^{\max}$ for all $t$ with $t' + k \cdot 7\Delta < t < t' + (k+1) \cdot 7\Delta$. Combining the inequalities above, it follows that there is a time $t^* > t'$ where $ets_{t^*}^{\max} = ets_{t^*}^{\min}$. In fact, $ets_t^{\max} = ets_t^{\min}$ infinitely often after time $t'$. Let $t''$ be the time after which all guild members wait at least $2 \cdot (\Delta + d)$ time units and at which $ets_{t''}^{\max} = ets_{t''}^{\min} = ets$. If no guild member enters a new epoch at time $t^*$ with $t'' \leq t^* \leq t'' + d$, then the lemma is proven. If some guild member starts a new epoch $ets + 1$ at time $t^*$, then $ets_{t^*}^{\min} = ets_{t^*}^{\max} + 1$. Therefore, by Lemma 17, $ets_{t^*+2\Delta}^{\min} = ets_{t^*+2\Delta}^{\max} = ets + 1$. Furthermore, by construction, no guild member complains about epoch $ets + 1$ at least up to time $t^* + 2\Delta + d$. Therefore, $ets_{t^{**}}^{\min} = ets_{t^{**}}^{\max} = ets + 1$ for $t^{**}$ in the range $t^* + 2\Delta \leq t^{**} \leq t^* + 2\Delta + d$. Since the guild members complain in increasing time intervals, the same argument can be applied to all $ets' \geq ets$, concluding the proof. □

**Theorem 19.** *Algorithm 4 implements the Asymmetric Epoch Change abstraction.*

*Proof. Monotonicity* is satisfied since the algorithm only alters the *ets* variable on line 69, where it increments it by one before triggering the *aec.newEpoch* event.

*Consistency* is also provided since the *leader* utility function is deterministic.

To show *Putch Resistance*, let $p_i$ be a wise process that has started a new epoch $ets > 0$. This means that $p_i$ has received a quorum $Q$ for itself of complaints for the previous epoch $ets - 1$ (line 68). By the consistency property of the quorum system, $Q \cap \mathcal{G}_{\max} \neq \emptyset$. Hence at least one member of $\mathcal{G}_{\max}$ has sent a complaint message for epoch $ets - 1$. Let $p_g$ be the first guild member to send a complaint message for epoch $ets - 1$. By Lemma 7 and the minimality of $p_g$, it cannot have sent the complaint message after satisfying the kernel condition on line 65. Therefore, $p_g$ must have triggered a complaint event (line 60), as this is the only other way it could send a complaint message.

Regarding *Eventual Progression*, when a kernel for all processes $K$, which is composed of correct processes, complains about a leader $p_i$ in epoch *ets*, then by Lemma 7, $K$ contains a guild member. This means, by *Eventual Entry*, that eventually every guild member enters epoch *ets*. Therefore every guild member eventually satisfies the kernel condition for epoch *ets* (line 65), after receiving all Complaint messages from $K$. Therefore, every guild member eventually sends a complaint message. Hence, by availability of the quorum system, every guild member eventually satisfies the quorum condition, causing them to start epoch $ets + 1$.

*Eventual Entry* has been proven in Lemma 15.

The *Unbounded Leadership* property has been shown in Lemma 18.

The *Uniform Leadership* property is satisfied since the *leader* utility function selects the leader in a round-robin fashion with respect to the epoch timestamp. □

## 6.2 Asymmetric Epoch Consensus

In this section, we introduce the epoch consensus primitive. First, we review the Practical Byzantine Fault Tolerance (PBFT) algorithm as presented by Cachin et al. [6] in the symmetric setting. We then explore an asymmetric version of PBFT proposed by Zanolini [23] and discuss its potential drawbacks. Finally, we propose our own asymmetric variant, called Asymmetric Practical Byzantine Fault Tolerance (APBFT).

### 6.2.1 Review of PBFT

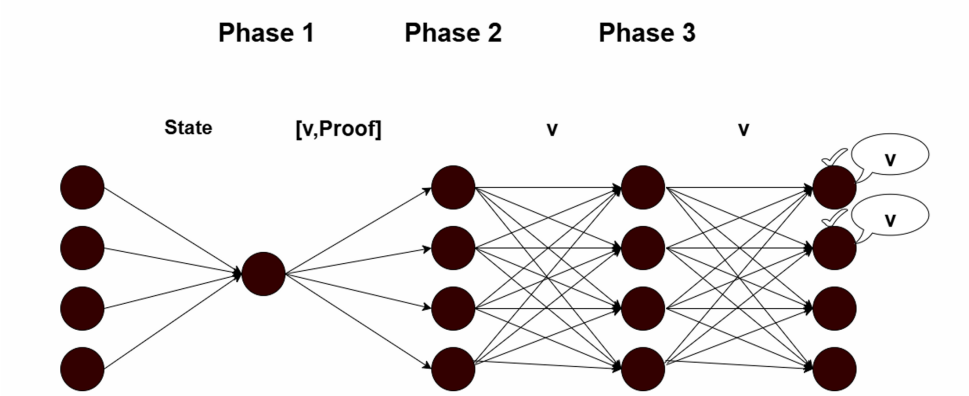



**Figure 6.1.** Communication flow of the PBFT algorithm as it progresses through its three phases.

We now review the PBFT protocol. First, we introduce the epoch consensus abstraction for the symmetric trust setting that it implements, and then we discuss the specifics of the protocol's implementation.

#### 6.2.1.1 Symmetric Epoch Consensus Specification

The epoch consensus abstraction is a primitive by which processes agree on a value proposed by the leader of a given epoch. Each epoch has a globally known leader, identified by its epoch timestamp. Since the leader may be faulty, an epoch consensus instance can be aborted, in which case the protocol moves to the next epoch under a new leader. An epoch terminates at a process either by triggering the *Decide* event or when a new epoch begins and the *StartEpoch* event is triggered.

Module 6 defines the properties of the Symmetric Epoch Consensus abstraction. These properties closely correspond to those of Weak Byzantine Consensus. The *Weak Validity* property allows decisions on values proposed in earlier epochs. The additional *Lock-in* property ties together the sequence of epochs, ensuring that once a correct process decides a value in any epoch, no correct process can decide a different value in any later epoch.

Module 6 omits termination conditions, as they must be adapted to the epoch change abstraction described above. Termination is revisited when we present the APBFT algorithm.

#### 6.2.1.2 Algorithm Description

The PBFT protocol proceeds in three phases, each motivated by the need to guard against Byzantine misbehavior and to ensure that future leaders can reliably detect past support:

**Module 6** Interface and Properties of the Symmetric Epoch Consensus Abstraction

**Module:**
Name: SymmetricEpochConsensus, **instance** *sep*

**Events:**
**Request**: *sep.Propose*$(v)$: Propose value $v$ for epoch consensus. Executed only by the leader $\ell$.
**Indication**: *sep.Decide*$(v)$: Output the decided value $v$ of epoch consensus.

**Request**: *sep.StartEpoch*$(ts, \ell)$: Start a new epoch *ts* with leader $\ell$.

**Properties:**
**EP1:** *Weak Validity*: If all processes are correct and a process decides $v$ in epoch *ets*, then $v$ was proposed by the leader of some epoch $ts' \leq ets$.

**EP2:** *Agreement*: No two correct processes decide differently during an epoch.

**EP3:** *Integrity*: Every correct process decides at most once during an epoch.

**EP4:** *Lock-in*: If a correct process has decided some value $v$ in an epoch with timestamp $ts' < ets$, then no correct process decides a different value in epoch *ets*.

---

1. **Proposal Phase.** The leader of epoch *ets* gathers from a quorum of replicas:

   - their *write-sets*, containing all values they have validated in earlier epochs, and
   - their persisted state $(ts, v)$, the most recent epoch and value each replica has recorded.

   Inspecting the highest-timestamped state $(ts, v)$ allows the leader to respect the lock-in rule, adopting any previously committed value rather than inventing a new one. The write-sets serve to certify that the chosen $(ts, v)$ was genuinely stored by correct replicas and cannot be fabricated by a Byzantine leader. The leader then broadcasts

   $$[\text{DECISION}, v, \textit{proof}]$$

   indicating that evidence for $v$ accompanies its proposal. The exact construction of *proof* is discussed in the next paragraph.

2. **Validation (Write) Phase.** Upon receiving $[\text{DECISION}, v, \textit{proof}]$, each replica:

   - Verifies that the leader supplied a valid proof.
   - Records $(ets, v)$ in its local write-set to persist the intent to decide $v$. This step prevents future leaders from fabricating a decision that never had quorum support.
   - Broadcasts $[\text{WRITE}, v]$ to all peers, echoing the proposed value to validate that the leader has sent the same decision value to all replicas.

   Once a replica collects a quorum of matching $[\text{WRITE}, v]$ messages, it considers $v$ *validated* and advances to the pre-commit phase.

3. **Pre-Commit Phase.** To ensure that any subsequent leader can detect that a quorum of replicas was ready to decide $v$, each replica:

   - Persists the pair $(ets, v)$ in its local state. This durable storage guarantees that future leaders cannot ignore the fact that $v$ had sufficient support.
   - Broadcasts $[\text{PRECOMMIT}, v]$ to its peers, signaling its final readiness to decide.

   When a replica gathers a quorum of identical $[\text{PRECOMMIT}, v]$ messages, it knows that enough correct replicas have durably recorded $(ets, v)$ and safely triggers

   $$\textit{sep.Decide}(v).$$

The communication flow and the 3 phase structure of the PBFT algorithm is illustrated in Figure 6.1

**Safety Mechanism.** The structure of phase 2 and 3 of the algorithm ensures that the persisted states and write-sets of the replicas contain all the information the leader of an epoch requires to avoid violating the *Lock-In* property. For any given epoch, if a correct process has decided in a previous epoch, then, by the structure of the third phase, a quorum of processes must have persisted that value to their state in that epoch. Therefore, by the quorum intersection property, each quorum of collected states must contain at least one such state testifying to a potential decision on that value. Therefore, in the absence of Byzantine processes, it would thus suffice for the leader to set its decision value to the value of the state with the highest timestamp. This is formalized using the predicate *couldbind*, which is defined below.

We adopt a fixed naming convention across all predicate definitions: the parameters *val* and *ts* always denote a value and a timestamp that together represent a state, and $S$ always denotes a set of tuples containing at least the components $(\mathit{process}, \mathit{value}, \mathit{timestamp})$. For each process *process*, the pair $(\mathit{value}, \mathit{timestamp})$ records its state at the beginning of the epoch. We refer to any such $S$ as a set of states.

In the following predicate definitions, the notation $S_{(\mathit{process})}$ denotes the projection of the set of tuples $S$ onto the *process* component of the tuples. To make the notation more concise: given a set $A$ consisting of tuples with components $(x, y, z)$, the notation $A_{(x,y)}$ denotes the set $\{(x, y) \mid (x, y, z) \in A\}$. This notation will be used throughout the text.

$\mathit{couldbind}(S, \mathit{ts}, \mathit{val}) \iff$

$S_{(\mathit{process})} \in \mathcal{Q}$ // The processes of $S$ form a quorum

$\wedge\ (\mathit{val}, \mathit{ts}) \in S_{(\mathit{value},\mathit{timestamp})}$ // The state $(\mathit{val}, \mathit{ts})$ is contained in the states of $S$

$\wedge\ \forall \mathit{ts}' \in S_{(\mathit{timestamp})} : \mathit{ts}' \leq \mathit{ts}$ // The state $(\mathit{val}, \mathit{ts})$ is the state with the highest timestamp in $S$

$\wedge\ \forall (\mathit{val}', \mathit{ts}') \in S_{(\mathit{value},\mathit{timestamp})} : \mathit{ts}' = \mathit{ts} \implies \mathit{val}' = \mathit{val}$ // The state $(\mathit{val}, \mathit{ts})$ is the unique state with maximal timestamp

In the presence of Byzantine processes, the *couldbind* predicate alone is no longer sufficient to guarantee the *Lock-In* property. After a process has decided on some value $v$, Byzantine processes can attempt to violate the *Lock-In* property by sending a forged state to the leader with a different decision value and a higher timestamp.

To prevent this attack, the leader and replicas only accept a state $(\mathit{val}, \mathit{ts})$ that satisfies *couldbind* if there is also evidence that *val* was written by a correct process in some epoch $\mathit{ts}' \geq \mathit{ts}$. Hence, if correct processes only write decision values that preserve the *Lock-In* property, this attack becomes impossible.

This is where the write-set comes into play: it records which decision values each process has actually written in each epoch. We formalize this verification using the *cert* predicate. Intuitively, the *cert* predicate checks whether the write-sets of a coreset of processes contain a write of the decision value *val* in an epoch during or after epoch *ts*. By the definition of a coreset, the *cert* predicate can only be satisfied if a correct process has indeed written the value as required.

Formally, the predicate takes as input a new type of argument, $W$, called a set of witnesses. $W$ consists of tuples containing at least the components $(\mathit{process}, \mathit{writeset})$, where each *writeset* is the write-set of process *process* at the start of the epoch. As with the set $S$, whenever we refer to a set $W$ for a predicate, it is assumed to be a set of tuples of this form.

$\mathit{cert}(W, \mathit{ts}, \mathit{val}) \iff$

$W_{(\mathit{process})} \in \mathcal{C}$ // The processes of $W$ form a coreset

$\wedge\ \forall \mathit{ws} \in W_{(\mathit{writeset})}, \exists \mathit{ts}' \geq \mathit{ts} : (\mathit{val}, \mathit{ts}') \in \mathit{ws}$ // The write-sets record that *val* was written in or after epoch *ts*

The predicate *bind* combines the *couldbind* and *cert* predicates to implement the safety mechanism described above.

$\mathit{bind}(S, W, \mathit{ts}, \mathit{val}) \iff$

$\mathit{couldbind}(S, \mathit{val}, \mathit{ts})$ // The state $(\mathit{val}, \mathit{ts})$ satisfies *couldbind*

$\wedge\ \mathit{cert}(W, \mathit{ts}, \mathit{val})$ // The set of witnesses $W$ certifies the state $(\mathit{val}, \mathit{ts})$

To protect against faulty leaders, replicas verify the leader's work by accepting its decision value only if it provides them with a set of states and witnesses that satisfy the *bind* predicate for that value. To prevent the leader from manipulating the collected states, each process signs the state it send to the leader, and replicas only accept states that are presented with correct signatures.

Finally, the leader and replicas need a way to handle the situation where none of the quorums satisfy *bind*. This situation occurs in particular after the initialization of the system, when every process has the state $(\perp, 0)$.

By disallowing $\perp$ as a decision value, collecting a quorum of initial states proves to the replicas that no correct process has decided in any previous epoch. Therefore, in this scenario, it is safe for the replicas to accept any decision value proposed by the leader without violating the *Lock-In* property. This condition is expressed by the predicate *unbound*.

$\mathit{unbound}(S) \iff$

$S_{(\mathit{process})} \in \mathcal{Q}$ // The processes of $S$ form a quorum

$\wedge\, \forall (\mathit{val}, \mathit{ts}) \in S_{(\mathit{value},\mathit{timestamp})} : (\mathit{val}, \mathit{ts}) = (\perp, 0)$ // Every process in $S$ has the initial state

The complete code for the epoch-consensus mechanism of the PBFT algorithm is shown in Algorithms 5–7. Every message is timestamped with the epoch, to ensure that messages from different epochs cannot interfere.

### 6.2.2 PAPBFT: A Hybrid Approach

In this section, we review an asymmetric PBFT epoch-consensus algorithm proposed by Zanolini [23]. We discuss the adaptations made to the original PBFT algorithm and highlight the main drawback of this variation. No formal specification of the algorithm's interface is given, since this algorithm is not the main focus of the section and is very similar to the asymmetric epoch-consensus interface of APBFT, which is presented in the next subsection.

#### 6.2.2.1 Algorithm Description

The usual recipe of replacing quorums, kernels, and core sets with their asymmetric counterparts [1] fails in phase 1 of the PBFT algorithm. In the first phase, the leader and replicas rely on states from a quorum of processes to deduce the system's state, in particular to identify whether a process has or has not decided in a previous epoch. In the symmetric setting, all processes share the same set of quorums; hence, each quorum serves as a snapshot of the system that all processes agree upon and accept. However, in the asymmetric setting, every process defines its own set of quorums, and therefore no single quorum exists that serves as a global snapshot of the system for all processes.

Zanolini works around this issue by operating on the quorum system $\mathcal{H}$, derived from the tolerated system $\mathcal{T}$ of the asymmetric quorum system. This resolves the problem, since the tolerated system is again a symmetric quorum system, and therefore its quorums once again serve as a global snapshot shared among the processes.

This change in the algorithm is reflected in a straightforward adjustment of the predicates *couldbind*, *cert*, and *unbound*, where each reference to the quorum system $\mathcal{Q}$ is replaced by the quorum system $\mathcal{H}$, and every reference to the coreset system $\mathcal{C}$ of $\mathcal{Q}$ is replaced by the coreset system $\mathcal{C}_{\mathcal{H}}$ of $\mathcal{H}$, while all other aspects of the predicates remain unchanged. The redefinition of the affected predicates is given below.

$\mathit{cert}(W, \mathit{ts}, \mathit{val}) \iff$

$W_{(\mathit{process})} \in \mathcal{C}_{\mathcal{T}}$ // The processes of $W$ form a coreset

$\wedge\, \forall \mathit{ws} \in W_{(\mathit{writeset})}, \exists \mathit{ts}' \geq \mathit{ts} : (\mathit{val}, \mathit{ts}') \in \mathit{ws}$ // The write-sets record that *val* was written in or after epoch *ts*

$\mathit{couldbind}(S, \mathit{ts}, \mathit{val}) \iff$

$S_{(\mathit{process})} \in \mathcal{H}$ //The processes of $S$ form a quorum

$\wedge\, (\mathit{val}, \mathit{ts}) \in S_{(\mathit{value},\mathit{timestamp})}$ // The state $(\mathit{val}, \mathit{ts})$ is contained in the states of $S$

$\wedge\, \forall \mathit{ts}' \in S_{(\mathit{timestamp})} : \mathit{ts}' \leq \mathit{ts}$ // The state $(\mathit{val}, \mathit{ts})$ is the state with the highest timestamp in $S$

$\wedge\, \forall (\mathit{val}', \mathit{ts}') \in S_{(\mathit{value},\mathit{timestamp})} : \mathit{ts}' = \mathit{ts} \implies \mathit{val}' = \mathit{val}$ // The state $(\mathit{val}, \mathit{ts})$ is the unique state with maximal timestamp

**Algorithm 5** PBFT/PAPBFT (Part 1, Decision Phase) (Process $p_i$).

**state**
73: $(val, ts, ws) \leftarrow (\perp, 0, \{\})$ // Process state
74: *proposal* $\leftarrow \perp$ // Processes proposal
75: *ets* $\leftarrow 0$ // Current epoch
76: $\ell \leftarrow \perp$ // Leader of the current epoch

// Phase 1 variables
77: *received* $\leftarrow \{\}$ // Collected states by the leader; stores tuples of the form (*timestamp*, *value*, *writeset*, *process*, *signature*), where *timestamp*, *value* and *writeset* is the local state of processes *process* and *signature* is its signature for that state

// Phase 2 variables
78: *sent* $\leftarrow$ FALSE // Tracks if the process has already sent a WRITE msg
79: *written* $\leftarrow [\perp]^n$ // Received writes

// Phase 3 variables
80: *decided* $\leftarrow$ FALSE // Tracks if the process has decided
81: *wrote* $\leftarrow$ FALSE // Tracks if the process has written to its state
82: *accepted* $\leftarrow [\perp]^n$ // Received decides

83: **upon event** *StartEpoch*(*epoch*, *leader*) **do**
84: // Reset the epoch specific state variables
85: *ets* $\leftarrow$ *epoch*
86: $\ell \leftarrow$ *leader*
87: *received* $\leftarrow \{\}$
88: *sent* $\leftarrow$ FALSE
89: *written* $\leftarrow [\perp]^n$
90: *decided* $\leftarrow$ FALSE
91: *wrote* $\leftarrow$ FALSE
92: *accepted* $\leftarrow [\perp]^n$

93: *prevState* $\leftarrow (val, ts, ws)$
94: $\sigma \leftarrow sign_i(prevState)$
95: send message [INPUT, (*prevState*, $\sigma$, *ets*)] to leader $\ell$

96: **upon event** *propose*($v$) **do**
97: **if** *proposal* $= \perp$ **then** *proposal* $\leftarrow v$
98: *proposed* $\leftarrow$ TRUE

99: **upon** receive a message [INPUT, (*prevState*, $\sigma$, *ts*$'$)] from $p_j$ **such that** *ts*$' =$ *ets* **do**
100: **if** $verify_j(prevState, \sigma)$ **then**
101: *received* $\leftarrow$ *received* $\cup \{(prevState^*, p_j, \sigma)\}$

102: **upon event** *proposed* $\wedge\ \exists S, W \subseteq$ *received*, $v \neq \perp$ *ts* $\in \mathbb{N}$ : *bind*($v$, *ts*, $S$, $W$) **do**
103: send message [BIND, $v$, *ts*, $S$, $W$, *ets*] to $p_j \in \mathcal{P}$

104: **upon event** *proposed* $\wedge\ \exists S \subseteq$ *received* : *unbound*($S$) **do**
105: send message [UNBOUND, *proposal*, $S$, *ets*] to $p_j \in \mathcal{P}$

106: **upon** receive a message [BIND, *val*, *ts*, $S$, $W$, *ts*$'$] from $\ell$ **such that** *ts*$' =$ *ets* **do**
107: **if** $\forall (state, p_j, \sigma) \in S \cup W$ : $verify_j(state, \sigma) \wedge bind(val, ts, S, W)$ **then**
108: **trigger** *Phase2*(*val*, *ets*)

109: **upon** receive a message [UNBOUND, *val*, $S$, *ts*$'$] from $\ell$ **such that** *ts*$' =$ *ets* **do**
110: **if** $\forall (state, p_j, \sigma) \in S$ : $verify_j(state, \sigma) \wedge unbound(S)$ **then**
111: **trigger** *Phase2*(*val*, *ets*)

**Algorithm 6** PBFT (Part 2, Write/Validation phase) (Process $p_i$).

112: **upon event** *Phase2*$(v, ts')$ **such that** $\neg$*written* $\wedge$ $ts' = ets$ **do**
113: $written \leftarrow$ TRUE
114: **if** $\exists ts' : (ts', v) \in ws$ **then**
115: $ws \leftarrow ws \setminus (ts', v)$
116: $ws \leftarrow ws \cup (ets, v)$
117: send message $[\text{WRITE}, v, ets]$ to all $p_j \in \mathcal{P}$

118: **upon** receive a message $[\text{WRITE}, v, ts']$ from $p_j$ **such that** $ts' = ets$ **do**
119: $written[j] \leftarrow v$

120: **upon** $\exists v \neq perp$ **such that** $\{p_j \in \mathcal{P} | written[j] = v\} \in \mathcal{Q}$ **do**
121: **trigger** *Phase3*$(v, ets)$

**Algorithm 7** PBFT (Part 3, Pre-Commit Phase) (Process $p_i$).

122: **upon event** *Phase3*$(v, ts')$ **such that** $ts' = ets$ **do**
123: $(val, ts) \leftarrow (v, ets)$
124: send message $[\text{PRECOMMIT}, v, ets]$ to $p_j \in \mathcal{P}$

125: **upon** receive a message $[\text{PRECOMMIT}, v, ts']$ from $p_j$ **such that** $ts' = ets$ **do**
126: $accepted[j] \leftarrow v$

127: **upon** $\neg decided \wedge \exists v \neq \perp$ **such that** $\{p_j \in \mathcal{P} | accepted[j] = v\} \in \mathcal{Q}$ **do**
128: $decided \leftarrow$ TRUE
129: **trigger** *sep.Decide*$(v)$

*unbound*$(S)$ $\iff$

$S_{(process)} \in \mathcal{H}$ // The processes of $S$ form a quorum

$\wedge \forall (val, ts) \in S_{(value,timestamp)} : (val, ts) = (\perp, 0)$ // Every process in $S$ has the initial state

Algorithms 8 and 9 show the implementation of phases two and three of the consensus algorithm. They differ from the symmetric algorithm only in that they replace the symmetric quorums with asymmetric ones. The pseudocode for the first phase is not shown as it is identical to that of the symmetric case (Algorithm 5), using the redefined *bind* and *unbound* predicates.

**Algorithm 8** PAPBFT (Part 2, Write/Validation Phase) (Process $p_i$).

130: **upon event** *Phase2*$(v, ts')$ **such that** $\neg$*written* $\wedge$ $ts' = ets$ **do**
131: $written \leftarrow$ TRUE
132: **if** $\exists ts' : (ts', v) \in ws$ **then**
133: $ws \leftarrow ws \setminus (ts', v)$
134: $ws \leftarrow ws \cup (ets, v)$
135: send message $[\text{WRITE}, v, ets]$ to all $p_j \in \mathcal{P}$

136: **upon** receive a message $[\text{WRITE}, v, ts']$ from $p_j$ **such that** $ts' = ets$ **do**
137: $written[j] \leftarrow v$

138: **upon** $\exists v \neq \perp$ **such that** $\{p_j \in \mathcal{P} | written[j] = v\} \in \mathcal{Q}_i$ **do**
139: **trigger** *Phase3*$(v, ets)$

#### 6.2.2.2 Drawback

This implementation has one major drawback. From a theoretical standpoint, it is a rather unsatisfactory solution to the issue encountered in the first phase when considering asymmetric quorum systems. The algorithm does not

**Algorithm 9** PAPBFT (Part 3, Pre-Commit Phase) (Process $p_i$).

140: **upon event** *Phase3*$(v, ts')$ **such that** $ts' = ets$ **do**
141: $(val, ts) \leftarrow (v, ets)$
142: send message $[\text{PRECOMMIT}, v, ets]$ to $p_j \in \mathcal{P}$

143: **upon** receive a message $[\text{PRECOMMIT}, v, ts']$ from $p_j$ **such that** $ts' = ets$ **do**
144: $accepted[j] \leftarrow v$

145: **upon** $\neg decided \wedge \exists v \neq \perp$ **such that** $\{p_j \in \mathcal{P} | accepted[j] = v\} \in \mathcal{Q}_i$ **do**
146: $decided \leftarrow \text{TRUE}$
147: **trigger** *aep.Decide*$(v)$

provide a true solution to the lack of a global view shared by the processes but instead circumvents the issue by relying once again on a symmetric quorum system. This approach offers no insight into how to address this dilemma within the asymmetric setting itself.

### 6.2.3 Introducing APBFT

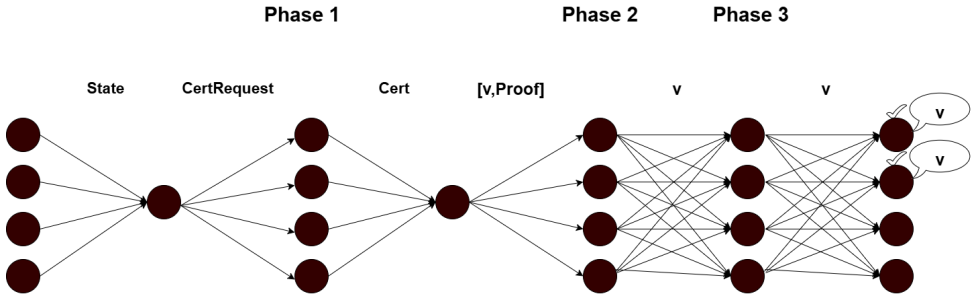



**Figure 6.2.** Communication flow of the APBFT algorithm as it progresses through its three phases. In contrast to the PBFT algorithm, the first phase includes an additional round of communication, in which the leader broadcasts certification requests and the replicas holding a certificate send back a response.

In this section, we introduce the APBFT algorithm, an asymmetric version of the PBFT algorithm that does not rely on the tolerated system. First, we discuss the adaptation of the epoch consensus interface to the asymmetric setting. Next, we describe the adaptations made to PAPBFT to derive the APBFT algorithm, and finally, we prove its correctness.

#### 6.2.3.1 Asymmetric Epoch Consensus Specification

We review the changes made to the epoch consensus interface (shown in Module 7) to adapt it to the asymmetric setting. The first four core properties of the epoch consensus abstraction remain semantically the same as in the symmetric setting. However, the guarantees of *Agreement* and *Lock-In* have been restricted to guild members.

The epoch consensus interface specifies two termination conditions. The first, called *Termination*, ensures that when a guild member is the leader of an epoch and the replicas give them sufficient time to fulfill their role, every guild member reaches consensus in that epoch. The second condition, called *One for All*, is a termination guarantee that ensures all guild members in an epoch eventually terminate if at least one other guild member has already decided.

This additional termination condition is required to handle cases where faulty leaders attempt to sabotage system liveness by allowing some guild members to terminate while blocking the progress of others. The *One for All* property prevents this from happening.

#### 6.2.3.2 Algorithm Description

As mentioned in the previous section, the main difficulty in reformulating the PBFT algorithm lies in how the leader and replicas share a global view of the system via a quorum. This shared view does not exist in the asymmetric setting. Recall what the replicas and the leader try to achieve in the first phase of the algorithm. The job of the

**Module 7** Interface and Properties of the Asymmetric Epoch Consensus Abstraction

**Module:**

Name: AsymmetricEpochConsensus, **instance** *aep*

**Events:**

**Setup**: *aep.StartEpoch*$(ets, \ell)$: Progresses epoch consensus to epoch *ets* with leader $\ell$.

**Request**: *aep.Propose*$(v)$: Proposes value $v$ for epoch consensus.

**Indication**: *aep.Decide*$(v)$: Output a decide value $v$ of epoch consensus.

**Properties:**

**EP1:** *Weak Validity*: If all processes are correct and a process decides $v$ in epoch *ets*, then $v$ was proposed by the leader of some epoch $ets' \leq ets$.

**EP2:** *Agreement*: No two wise processes decide differently in an epoch.

**EP3:** *Integrity*: Every correct process decides at most once in each epoch.

**EP4:** *Lock-in*: If a wise processes has decided some value $v$ in an epoch with timestamp $ts' < ts$, then no wise process decides a value different from $v$ in epoch *ts*.

**EP5:** *Termination*: If the leader $\ell$ of some epoch *ets* is a guild member, has proposed a value, and all guild members are simultaneously in epoch *ets* for a fixed time period, then every guild member decides in epoch *ets*.

**EP6:** *One for All*: In every execution with a guild, When a guild member decides in some epoch *ets*, and all guild members are simultaneously in epoch *ets* for a fixed amount of time, then every guild member decides in epoch *ets*.

leader is to identify a decision value that it can prove to a sufficient number of replicas does not violate the *Lock-In* property, based on the local states of the replicas. The job of the replicas is to verify that proof and proceed to phase 2 of the algorithm only with decision values that respect *Lock-In*.

The key to circumventing the issue of no shared global quorum to rely on is for the leader to collect, for each process, one of its personal quorums to provide it with a personalized proof that the decision value is safe to accept.

In the symmetric setting, the *bind* predicate was used to identify decision values that respect *Lock-In*. The justification for why the *bind* predicate preserves *Lock-In* generalizes,for guild members, to the asymmetric setting when swapping symmetric quorums with asymmetric ones.

Below, the redefinition of the *bind* and *unbound* predicates is given. We defer the redefinition of the *cert* predicate to the end of this section, since the certification mechanism needs to be adapted due to weaker guarantees of the epoch change primitive in the asymmetric setting. However, the asymmetric *cert* predicate preserves its properties for guild members and serves the same purpose as it does in the symmetric setting.

The predicates additionally take a process identifier as input to index the individual quorum systems. The parameters of the predicates remain unchanged; only the structure of the set of witnesses $W$ differs, which will be discussed when presenting the redefinition of *cert*.

$\mathit{bind}_i(S, W, \mathit{val}, \mathit{ts}) \quad \Longleftrightarrow$

$\mathit{couldbind}_i(S, \mathit{val}, \mathit{ts})$ // The state $(\mathit{val}, \mathit{ts})$ satisfies $\mathit{couldbind}_i$

$\wedge\ \mathit{cert}_i(W, \mathit{ts}, \mathit{val})$ // The set of witnesses $W$ certifies the state $(\mathit{val}, \mathit{ts})$

$\mathit{couldbind}_i(S, \mathit{val}, \mathit{ts}) \quad \Longleftrightarrow$

$S_{(\mathit{process})} \in \mathcal{Q}_i$ // The processes of $S$ form a quorum for process $p_i$

$\wedge\ (\mathit{val}, \mathit{ts}) \in S_{(\mathit{value},\mathit{timestamp})}$ // The state $(\mathit{val}, \mathit{ts})$ is contained in the states of $S$

$\wedge\ \forall \mathit{ts}' \in S_{(\mathit{timestamp})} : \mathit{ts}' \leq \mathit{ts}$ // The state $(\mathit{val}, \mathit{ts})$ is the state with the highest timestamp in $S$

$\wedge\ \forall (\mathit{val}', \mathit{ts}) \in S_{(\mathit{value},\mathit{timestamp})} : \mathit{val}' = \mathit{val}$ // The state $(\mathit{val}, \mathit{ts})$
is the unique state with maximal timestamp

This adapted *bind* predicate behaves identically as in the symmetric case for guild members. In particular, after a guild member has decided on some value $v$, for every guild member $p_i$, $\mathit{bind}_i$ can only be satisfied by the decision value $v$. However, the converse is not true: *bind* being satisfied does not guarantee that a guild member has previously decided. Recall that in the symmetric case, the *unbound* predicate proved to a replica that no correct process had yet decided. This concept generalizes naturally to the asymmetric setting.

$\mathit{unbound}_i(S) \quad \Longleftrightarrow$

$S_{(\mathit{process})} \in \mathcal{Q}_i$ // The processes of $S$ form a quorum for process $p_i$

$\wedge\ \forall (\mathit{val}, \mathit{ts}) \in S_{(\mathit{value},\mathit{timestamp})} : (\mathit{val}, \mathit{ts}) = (\bot, 0)$ // Every process in $S$ has the initial state

The key idea in the construction of the APBFT algorithm is the introduction of an additional predicate that proves to a guild member that no guild member has yet decided. By combining the ideas of the *unbound* and *bind* predicates, a replica can conclude that a decision value is safe to accept when confronted with a set of states and witnesses $S$ and $K$ that bind to a state $(\mathit{val}, \mathit{ts})$, and, in addition, a set of witnesses $K'$ that certify they have written some different value $\mathit{val}' \neq \mathit{val}$ in an epoch with timestamp $\mathit{ts}' > \mathit{ts}$.

$\mathit{debind}_i(S, W, W', \mathit{val}, \mathit{ts}) \quad \Longleftrightarrow$

$\exists \mathit{val}' \neq \mathit{val}, \mathit{ts}' < \mathit{ts} : \mathit{bind}_i(S, W, \mathit{val}', \mathit{ts}')$ // Predicate $\mathit{bind}_i$ is satisfied
for a value $\mathit{val}' \neq \mathit{val}$ with a timestamp $\mathit{ts}' < \mathit{ts}$

$\wedge\ \mathit{cert}_i(W', \mathit{val}, \mathit{ts})$ // The set of witnesses $W'$ certifies $(\mathit{val}, \mathit{ts})$

This third predicate increases the number of decision values a replica can accept, thereby facilitating the leader's task of identifying a decision value that a sufficient number of replicas can accept.

The final decision rule, by which a replica can identify safe decision values, is expressed by the predicate $V$ (for Verify). It combines the three predicates: *bind*, *unbound*, and *debind*.

$V_i(S, W, W', val) \quad \Longleftrightarrow$

$\quad$ *unbound*$_i(S)$ // Predicate *unbound*$_i$ is satisfied

$\quad \vee \exists ts \in \mathbb{N} :$ *bind*$_i(S, W, val, ts)$ // Predicate *bind*$_i$ is satisfied

$\quad \vee \exists ts \in \mathbb{N} :$ *debind*$_i(S, W, W', val, ts)$ // Predicate *debind*$_i$ is satisfied

The predicate $V$ has the desirable property that, in every execution of the algorithm, it can be satisfied with the same decision value for every guild member. Thus, for the leader of an epoch to choose a suitable decision value, it must collect states until it detects a quorum for itself of processes that satisfy $V$ with the same decision value. If the leader is a guild member, such a quorum is guaranteed to exist, and this quorum suffices to drive the epoch to consensus.

The decision rule of the leader is formalized as the predicate $C_i(Q, S, W, val)$, standing for Collect. The input parameter $Q$ is a set of processes forming a quorum for the leader $p_i$. The sets $S$ and $W$ are the sets of states and witnesses, respectively. They carry all the required information for satisfying the $V$ predicate for every process in the quorum $Q$ for the same decision value *val*.

$C_i(Q, S, W, val) \quad \Longleftrightarrow$

$\quad Q \in \mathcal{Q}_i$ // The set of processes $Q$ is a quorum for the leader

$\quad \wedge \forall p_j \in Q, \exists S' \subseteq S,\ W', W'' \subseteq W, ts \in \mathbb{N} : V_j(S', W', W''val, ts)$
$\quad\quad$ // For every process in $Q$ the process can verify *val*

Since the leader can only determine a decision value for a quorum for itself, processes also write a value in the second phase of the algorithm when they receive a kernel for themselves of write messages from their peers. This ensures that eventually all wise processes write a value, even if they never complete the first phase.

One final change must be made to the original PBFT algorithm due to the weaker properties of the Asymmetric Epoch Change abstraction. Namely, in the asymmetric setting, the proposed epoch change mechanism, *Eventual Entry*, only ensures progression through the epochs for guild members. This means that wise and naive processes may get stuck in some epoch, even though the system did not, and will not, reach consensus in that epoch. This is problematic because the $V$ predicate relies on kernels of processes that certify a state with their write sets.

When a wise process decides on $v$ in some epoch *ets*, the structure of the second and third phases of the algorithm ensures that, for every guild member, a kernel for itself of correct processes exists that certifies the state $(v, ets)$. However, since this kernel need not consist exclusively of guild members, some processes certifying the state may get stuck in lower epochs. As a result, the leader may never collect their states and thus never observe the kernel, potentially jeopardizing the liveness of the algorithm.

To address this issue, instead of processes sending their write-sets to the leader at the start of the first phase of the algorithm, they only send their local state $(val, ts)$. The leader then uses the *couldbind* predicate to identify states that require certification. The leader subsequently broadcasts a request to the replicas, asking them to certify the requested state. Once a process is able to certify the requested state, it sends a signed acknowledgment back to the leader, attesting that it can indeed certify the requested state. This additional communication step is illustrated in Figure 6.2

The leader stores this information in a set of tuples called *witnesses*, of the form $(value, timestamp, witnessSet)$, where *witnessSet* is a set of tuples of the form $(process, signature)$, each consisting of a process and a signature. These tuples represent processes that have messaged the leader, stating that they can certify the state $(value, timestamp)$, with the signature preventing the leader from forging certificates. The *cert*$_i$ predicate is redefined to operate on this new data structure; it checks whether a set of witnesses $W$ contains a kernel for process $p_i$ that certify *val* for some timestamp $ts' \geq ts$.

In the symmetric setting, we relied on coresets for the certification process. In the APBFT algorithm, we instead rely on kernels, since kernels provide the same key guarantee as coresets in the symmetric setting, namely, that they contain at least one correct process, which is required to justify the definition of the *cert* predicate.

However, in the asymmetric setting, coresets provide weaker guarantees than kernels. Coresets for guild members still preserve the guarantee that they contain at least one correct process. This guarantee is weaker, however, because this correct process might be naive, making its testimony that it has written a value unreliable, as naive processes are not protected from writing decision values that violate the *Lock-In* property. Kernels for guild members, on the other hand, provide the stronger guarantee of containing at least one other guild member, which mimics the mechanism of coresets in the symmetric setting.

$\mathit{cert}_i(W, \mathit{ts}, \mathit{val}) \iff$

$\exists K \in \mathcal{K}_i, \forall p_j \in K, \exists (\mathit{val}, \mathit{ts}', w) \in W : \mathit{ts}' \geq \mathit{ts} \wedge p_j \in w_{(\mathit{process})}$
// $W$ contains a kernel of certificates for $(\mathit{ts}, \mathit{val})$

The final code for the first and second phases is shown in Algorithms 10–12. Phase 3 remains unchanged as in Algorithm 9.

#### 6.2.3.3 Algorithm Correctness

In this section, we prove the correctness of the APBFT algorithm. We begin by formalizing the properties of the predicates discussed in the previous section and conclude by showing that APBFT implements the Asymmetric Epoch Consensus abstraction.

Throughout the following proofs, we make repeated use of the following terminology: In the execution of the epoch consensus algorithm, the first *startEpoch* event marks the beginning of the first epoch. The period between two *startEpoch* events defines the execution of an epoch, where the first event is referred to as the start of the epoch and the second as the end. Each epoch is identified by the value of the variable *ets*.

A process $p_i$ is said to satisfy a predicate if, during the execution of the protocol, the predicate evaluates to TRUE at process $p_i$ at some time $t$.

The following lemma establishes the desired property of the *cert* predicate.

**Lemma 20.** *In every execution with a guild, if a correct process $p_j$ satisfies the predicate $\mathit{cert}_i(W, v, \mathit{ts})$ in epoch ets at time t, where $p_i \in \mathcal{G}max$, then there exists another process $p_g \in \mathcal{G}_{max}$ that wrote $v$ at time $t_g < t$ in some epoch $\mathit{ts}_g \geq \mathit{ts}$.*

*Proof.* A correct processes $p_j$ only satisfies $\mathit{cert}_i$ when it is the leader of an epoch (on lines 198 and 193), while satisfying predicates $C_j$ and $V_i$. Also $p_i$ satisfies $\mathit{cert}_i$ when satisfying $V_i$ on line 203. In all three cases, $\mathit{cert}_i$ is invoked on a set $W$ containing tuples $(v', \mathit{ts}', w)$, where $w$ contains process-signature pairs $(p_h, \sigma)$. In all three cases, on lines 191 and 202, before satisfying $\mathit{cert}_i$, the processes first check that the signature $\sigma$ was generated by process $p_h$ for the state $(v', \mathit{ts}')$ by verifying that $\mathit{verify}_h((v', \mathit{ts}'), \sigma)$ is true for all tuples in $W$. However, a correct processes $p_h$ only generates this signature on lines 186 and 212, after checking that its write set contains a pair $(v', \mathit{ts}'')$ with $\mathit{ts}' \geq \mathit{ts}''$. Since $p_h$ only adds tuple $(v', \mathit{ts}'')$ to its write set on line 209 when writing $v'$ in epoch $\mathit{ts}''$, this establishes that every correct process in $w$ has, in fact written $v'$ in some epoch $\mathit{ts}'' \geq \mathit{ts}'$ at some time $t' < t$ as desired.

Furthermore, by the definition of the predicate $\mathit{cert}_i$, $W$ satisfies it only when it contains a kernel $K \in \mathcal{K}_i$ for $p_i$. By Lemma 7, there is a guild member $p_g \in \mathcal{G}_{\text{max}}$ contained in $K$. Also, by the definition of $\mathit{cert}_i$ there is a tuple $(v, \mathit{ts}_g, w_g)$ in $W$ such that $p_g \in w_g$ and $\mathit{ts}_g \geq \mathit{ts}$. As argued above, this establishes that $p_g$ wrote $v$ in epoch $\mathit{ts}_g$ at some time $t_g < t$, as desired. □

**Lemma 21.** *If a correct process $p_h$ satisfies the predicate $\mathit{unbound}_i(S)$ or $\mathit{couldbind}_i(S, \cdot, \cdot)$ in some epoch ets, for a correct process $p_i$, then for all $(v, \mathit{ts}, p_j) \in S_{(\mathit{value},\mathit{timestamp},\mathit{process})}$, the pair $(v, \mathit{ts})$ is the local state of $p_j$ at the end of epoch ets $- 1$.*

*Proof.* The predicate $\mathit{unbound}_i$ or $\mathit{couldbind}_i$ is only satisfied on lines 181, 193, 198 and 203, either by the leader of epoch *ets* or by $p_i$ itself if its correct. In both cases, the set $S$ consists of tuples $(v, \mathit{ts}, p_j, \sigma)$ and the processes ensure that the state $(v, \mathit{ts})$ indeed originates from processes $p_j$ by performing the check $\mathit{verify}_j((v, \mathit{ts}), \sigma)$ on lines 179 and 201, before satisfying the predicates. Since a correct $p_j$ only generates this signature at the start of the first phase(line 173) this establishes that $(v', \mathit{ts}')$ is the state of $p_j$ at the end of epoch $\mathit{ets} - 1$. □

The next lemma establishes the property, that for guild members $p_i$ the predicate $V_i$ can only be satisfied with decision values that do not violate the *Lock-In* property, starting off with two helper lemmas.

**Lemma 22.** *If in some epoch ets a wise process $p_i$ writes $(v, \mathit{ets})$ to its local state, then no other wise process writes $(v', \mathit{ets})$ to its local state for $v' \neq v$ in that epoch.*

*Proof.* Process $p_i$ only writes $(v, \mathit{ets})$ to its local state after receiving WRITE messages for $v$ from a quorum $Q$ for itself (lines 219 and 123). Assume that there is another wise process that writes $(v', \mathit{ets})$ to its local state with $v \neq v'$, which it would only do after receiving WRITE messages for $v'$ from a quorum $Q'$ for itself (line 219). By the quorum intersection property, there is a correct process in the intersection of $Q$ and $Q'$, implying that it has sent a WRITE message for both $v$ and $v'$, contradicting the fact that a correct process only sends a single WRITE message in each epoch. This is ensured by the fact that the *write* event is guarded by a $\neg$*wrote* condition, which is set to TRUE before sending the write message on line 205. The *wrote* variable is only reset to FALSE when triggering the *startEpoch* event, which also increments the *ets* variable(line 161). □

**Algorithm 10** State Variables and Epoch Initialization of APBFT (Process $p_i$).

**implements**
AsymmetricEpochConsensus, **instance** *aep*

**state**
148: $(val, ts, ws) \leftarrow (\perp, 0, \{\})$ // Process state
149: $proposal \leftarrow \perp$ // Processes proposal
150: $ets \leftarrow 0$ // Current epoch
151: $\ell \leftarrow \perp$ // Leader of the current epoch

// Phase 1 variables
152: $decisionvalue \leftarrow \perp$ // Leaders identified decision value
153: $received \leftarrow \{\}$ // Collected states by the leader; stores tuples of the form
$(timestamp, value, process, signature)$, where *timestamp* and *value* is the local state
of processes *process* and *signature* is its signature for that state

// Phase 2 variables
154: $sent \leftarrow$ FALSE // Tracks if the process has already sent a WRITE msg
155: $written \leftarrow [\perp]^n$ // Received writes

// Phase 3 variables
156: $decided \leftarrow$ FALSE // Tracks if the process has decided
157: $wrote \leftarrow$ FALSE // Tracks if the process has written to its state
158: $accepted \leftarrow [\perp]^n$ // Received decides

// certification datastructures
159: $witnesses \leftarrow \{\}$ // Contains tuples of the form $(value, timestamp, certificates)$ where *certificates*
is a set of tuples of the form $(process, signature)$ of processes
that certify the state $(value, timestamp)$ and there signatures *signature*
160: $certify \leftarrow \{\}$ // Set of tuples of the form $(value, timestamp, process)$
storing the state pairs $(value, timestamp)$ that leader *process* needs a certificate for

161: **upon event** *StartEpoch*$(epoch, leader)$ **do**
162: // Reset the epoch specific state variables
163: $ets \leftarrow epoch$
164: $\ell \leftarrow leader$
165: $received \leftarrow \{\}$
166: $sent \leftarrow$ FALSE
167: $written \leftarrow [\perp]^n$
168: $decided \leftarrow$ FALSE
169: $wrote \leftarrow$ FALSE
170: $accepted \leftarrow [\perp]^n$
171: $decisionvalue \leftarrow \perp$

172: $prevState \leftarrow (val, ts)$
173: $\sigma \leftarrow sign_i(prevState)$
174: send message [INPUT, $(prevState, \sigma, ets)$] to leader $\ell$

**Algorithm 11** APBFT (part 1, Decision Phase) (Process $p_i$).

175: **upon event** *propose*$(v)$ **do**
176: **if** *proposal* $=\perp$ **then** *proposal* $\leftarrow v$
177: *proposed* $\leftarrow$ TRUE

178: **upon** receive a message [INPUT, $(prevState, \sigma, ts')$] from $p_j$ **such that** $ts' = ets$ **do**
179: **if** $verify_j(prevState, \sigma)$ **then**
180: $received \leftarrow received \cup \{(prevState^*, p_j, \sigma)\}$

181: **upon** $\exists p_j \in \mathcal{P},\ v \neq \perp,\ ts \in \mathbb{N}, S \subseteq received : couldbind_j(S, v, ts) \wedge (v, ts, \cdot) \notin witnesses$ **do**
182: $witnesses \leftarrow witnesses \cup \{(v, ts, \{\})\}$
183: send message [CERTIFY, $v, ts$] to all $p_j \in \mathcal{P}$

184: **upon** receive a message [CERTIFY, $v, ts$] from $p_j$ **do**
185: **if** $\exists ts' \leq ts : (v, ts') \in ws$ **then**
186: $\sigma \leftarrow sign_i((v, ts))$
187: send message [VERIFIED, $(v, ts), \sigma$] to $p_j$
188: **else**
189: $certify \leftarrow certify \cup \{(v, ts, p_j)\}$

190: **upon** receive message [VERIFIED, $(v, ts), \sigma$] from $p_j$ **do**
191: **if** $verify_j((v, ts), \sigma) \wedge \exists (v, ts, w) \in witnesses$ **then**
192: $w \leftarrow w \cup (p_j, \sigma)$

193: **upon** $proposal \neq \perp \wedge \exists v \neq \perp,\ Q \in \mathcal{Q}_i : C_i(Q, received, witnesses, v)$ **do** // Only leader $\ell$
194: **if** $\forall p_j \in Q, \exists S_j \subseteq received : unbound_j(S_j)$ **then**
195: $decisionvalue \leftarrow proposal$
196: **else**
197: $decisionvalue \leftarrow v$

198: **upon** $decisionvalue \neq \perp,\ \exists p_j \in \mathcal{P},\ S \subseteq received,\ W, W' \subseteq witnesses : V_j(S, W, W', decisionvalue)$ **do**
199: send message [BIND, $S, W, W', decisionvalue, ets$] to $p_j$

200: **upon** receive a message [BIND, $S, W, W', decisionvalue, ts'$] from $\ell$ **such that** $ts' = ets$ **do**
201: **if** $\forall (state, p_j, \sigma) \in S : verify_j(state, \sigma)$
202: $\wedge \forall (v, ts, w) \in W \cup W', (p_j, \sigma) \in w : verify_j((v, ts), \sigma)$
203: $\wedge V_i(S, W, W', decisionvalue)$ **then**
204: **trigger** *write*$(decisionvalue, ets)$

**Algorithm 12** APBFT (part 2, Write/Validation Phase) (Process $p_i$).

205: **upon event** *write*$(v, ts')$ **such that** $\neg wrote \wedge ts' = ets$ **do**
206: *wrote* $\leftarrow$ TRUE
207: **if** $\exists ts' : (ts', v) \in ws$ **then**
208: $ws \leftarrow ws \setminus (ts', v)$
209: $ws \leftarrow ws \cup (ets, v)$
210: send message [WRITE, $v, ets$] to all $p_j \in \mathcal{P}$
211: **if** $\exists ts \leq ets,\ p_j \in \mathcal{P} : (v, ts, p_j) \in certify$ **then**
212: $\sigma \leftarrow sign_i((v, ts))$
213: send message [VERIFIED, $(v, ts), \sigma$] to $q$
214: $certify \leftarrow certify \setminus \{(v, ts, p_j)\}$

215: **upon** $\exists v \neq \perp$ **such that** $\{p_j \in \mathcal{P} | written[j] = v\} \in \mathcal{K}_i$ **do**
216: **trigger** *write*$(v, ets)$

217: **upon** receive a message [WRITE, $v, ts'$] from $p_j$ **such that** $ts' = ets$ **do**
218: $written[j] \leftarrow v$

219: **upon** $\exists v \neq \perp$ **such that** $\{p_j \in \mathcal{P} | written[j] = v\} \in \mathcal{Q}_i$ **do**
220: **trigger** *Phase3*$(v, ets)$

**Lemma 23.** *After a wise process $p_j$ decides $v \neq \perp$ in some epoch ets, no correct process satisfies unbound$_i$ for any wise process $p_i$.*

*Proof.* By the structure of the third phase of the algorithm, process $p_j$ only decides $v$ after receiving PRECOMMIT messages for $v$ from a quorum $Q_j \in \mathcal{Q}_j$ for itself (line 127). Observe that a correct process only sends a PRECOMMIT message after writing $(v, ets)$ to its local state (line 123).

Now assume that there is a correct process $p_k$ that satisfies $unbound_i(S)$ in some epoch $ets' > ets$. By the quorum intersection property, $S_{(process)}$ and $Q_j$ intersect in a correct process $p_h$. Let $(v_h, ets_h, p_h, \cdot) \in S$ be the tuple in $S$ corresponding to process $p_h$. Since $unbound_h(S)$ is satisfied, $v_h = \perp$. By Lemma 21, $(v_h, ets_h)$ is the local state of $p_h$ at the end of epoch $ets' - 1$. Let $t$ be the time at which process $p_h$ writes $(v, ets)$ to its local state in epoch $ets$. Since $ets' - 1 \geq ets$ this means that processes $p_h$ ends epoch $ets' - 1$ at time $t' > t$. Observe that no correct process ever writes $\perp$ to its local state, which implies that $v_h \neq \perp$, a contradiction. □

**Lemma 24.** *In every execution with a guild, if a correct process satisfies the predicate $V_i(S, W, W', v)$ in some epoch ets and $p_i \in \mathcal{G}_{max}$, then no wise process has decided a value $v' \neq v$ in any epoch $ts' < ets$.*

*Proof.* For the sake of contradiction assume that $V_i(S, W, W', v)$ is satisfied, yet a wise process $p_w$ has decided $v' \neq v$ in some epoch $ts' < ets$. Process $p_w$ only decides after receiving PRECOMMIT messages for $v'$ from a quorum $Q_w$ for itself (line 127). Each correct process in $Q_w$ only send a PRECOMMIT message for $v'$ after setting its local state to $(v', ts')$ (line 123).

Since by Lemma 23, $unbound_i$ cannot be satisfied, $bind_i$ or $debind_i$ must be satisfied for $V_i$ to hold.

Observe that correct processes only write strictly increasing timestamps to their local state, since the sequence of epochs is strictly increasing. By definition, a timestamp $ts''$ can only satisfy $couldbind(S, v', ts'')$ if $ts''$ is the maximal timestamp in $S$. Therefore, by the quorum intersection property of $Q_w$ and $S_{(process)}$ and the above observation applied to $Q_w$, it follows that $ts''$ can only satisfy $couldbind_i$, if $ts'' \geq ts'$. Hence $bind_i$ can only be satisfied by a timestamp $\geq ts'$. Therefore, by definition, $debind_i$ can also only be satisfied by a timestamp $> ts'$.

In fact, like $debind_i(S, W, W', v, ts'')$, $bind_i(S, W, v, ts'')$ can also only be satisfied by a $ts''$, that is strictly larger than $ts'$. This also follows from the quorum intersection property of $S_{(process)}$ and $Q_w$, and the unique maximal state condition of the $couldbind_i$ predicate. In the case where $ts'' = ts'$ if, for sake of contradiction, $bind(S, W, v, ts')$ is satisfied, then $(v, ts')$ is the unique maximal state in $S$. By quorum intersection of $S_{(process)}$ and $Q_w$, $S$ contains the state information of a correct process $p_c \in Q_w$. Let $(v_c, ts_c, p_c, \cdot) \in S$ be the tuple in $S$ associated with process $p_c$. Since correct processes write strictly increasing timestamps to their local state, $ts_c \geq ts'$. Therefore, $ts_c$ is the maximal timestamp in $S$, and it follows that $ts_c = ts'$ and $v_c = v$. However, $p_c$ had state $(v', ts')$, and since $p_c$ only writes strictly increasing timestamps to its state, it follows that $v' = v$, a contradiction. This establishes that $bind_i$ can only be satisfied with a timestamp $> ts'$ for value $v$.

By definition for $bind_i(S, W, v, ts'')$ or $debind_i(S, W, W', v, ts'')$ to be satisfied with a timestamp $ts'' > ts'$, it must hold that $cert_i(W, v, ts'')$ or $cert_i(W', v, ts'')$ is satisfied. By Lemma 20, a guild member must exist that writes $v$ during or after epoch $ts''$. Let $p_g \in \mathcal{G}_{\text{max}}$ be the first guild member that writes $v$ in some epoch $ets_g$ such that $ets_g > ts'$. Denote the time of this event by $t_{g_1}$. Since $p_g$ is the first guild member to write $v$ in epoch $ets_g$, by Lemma 7, its write cannot have occurred by satisfying the kernel condition on line 215. Therefore $p_g$ must have written the value after returning it from the first phase (line 204). This means that $p_g$ satisfied $V_g(S_g, W_g, W'_g, v)$ for some $S_g$, $W_g$ and $W'_g$. Denote by $t_{g_2} < t_{g_1}$ the time at which this occurs. Since $ets_g > ts'$, by the same reasoning as above, $unbound_g(S_g)$ is unsatisfiable and $bind_g(S_g, W_g, v, ts_g)$ or $debind_g(S_g, W_g, W'_g, v, ts_g)$ can only be satisfied by a timestamp $ts_g > ts'$. This, as argued above, implies that another guild member $p_{g'} \in \mathcal{G}_{\text{max}}$ wrote $v$ in some epoch with timestamp $ets_{g'} > ts'$, at some time $t_{g'}$ where clearly $t_{g'} < t_{g_2} < t_{g_1}$, contradicting the minimality of $p_g$. □

Finally we establish the liveness property of the $C_i$ predicate showing that it is always satisfiable when the leader is a guild member, again starting off with a helper lemma.

**Lemma 25.** *when a wise process $p_w$ writes $(val, ets)$ to its local state at time $t$, then at any time $t' \geq t$, for every other wise process $p_{w'}$ a kernel $K \in \mathcal{K}_{w'}$ for $p_{w'}$ of correct processes exist that can verify the state $(val, ets)$. Meaning that every process in $K$ has an entry $(val, ts)$ in its write set, such that $ts \geq ets$.*

*Proof.* $p_w$ only writes $(val, ets)$ to its local state after receiving a WRITE message for *val* from a quorum $Q_w$ for itself in epoch *ets*. By lemma 6 $Q_w$ contains a kernel $K_{w'}$ for $p_{w'}$ of correct processes for every wise process $p_{w'}$. Since every correct process only sends a WRITE message for *val* in epoch *ets* after adding $(val, ets)$ to its write set. Furthermore a correct processes never removes a value from its write set, it only replaces a tuple $(val, ets)$ by another tuple $(val, ets')$, with $ets' > ets$, when writing *val* in a later epoch $ets'$. Hence every process in $K_{w'}$ can always verify $(val, ets)$ after time $t$. □

**Lemma 26.** *After GST, in every execution with a guild, if the leader $p_\ell \in \mathcal{G}_{max}$ of some epoch ets is a guild member and all guild members start epoch ets, then $p_\ell$ satisfies $C_\ell$ within $3\Delta$ time steps, assuming $p_\ell$ does not move to a new epoch in the meantime.*

*Proof.* After the last guild member starts epoch *ets* at time $t$, he sends his state to $p_\ell$ (line 161), which arrives no later than at time $t + \Delta$. Let $Q_\ell$ denote the leader's guild quorum, and let $Q_g$ denote the guild quorum for each guild member $p_g \in Q_\ell$. Once $p_\ell$ receives the state from the last guild member, the leader can satisfy $\mathit{couldbind}_g$ or $\mathit{unbound}_g$ for every guild member $p_g$.

Let $S^g$ be the states of the processes in $Q_g$. More precisely, define $S^g := \{(v, ts, p_j, \sigma) \in \mathit{received} \mid p_j \in Q_g\}$. If every state in $S^g$ is the initialization state, i.e., $\forall (v, ts) \in S^g_{(\mathit{value},\mathit{timestamp})} : (v, ts) = (\bot, 0)$, then $\mathit{unbound}_g(S^g)$ is satisfied. Observe that for all $ts > 0$, we have $(\bot, ts) \notin S^g_{(\mathit{value},\mathit{timestamp})}$, since correct processes never write $\bot$ to their local state. Hence if $\mathit{unbound}_g(S^g)$ is not satisfied, it follows that there exists a state $(v, ts) \in S^g_{(\mathit{value},\mathit{timestamp})}$ such that $v \neq \bot$. Moreover, since correct processes only write strictly increasing timestamps to their local state, it follows that $ts > 0$. Furthermore, since every process in $Q_g$ is wise, it follows from Lemma 22 that states in $S^g$ that share the same timestamp have the same decision value. More formally, this means that $\forall (v, ts), (v', ts') \in S^g_{(\mathit{value},\mathit{timestamp})} : ts = ts' \implies v = v'$. Hence, some state $(v_g^{\max}, ts_g^{\max})$ with a maximal timestamp in $S^g$ clearly satisfies $\mathit{couldbind}_g(S^g, v_g^{\max}, ts_g^{\max})$.

This establishes that at time $t + \Delta$, the leader can either satisfy $\mathit{unbound}_g$, or has satisfied $\mathit{couldbind}_g$ and broadcast a verification request (line 181) for the state $(v_g^{\max}, ts_g^{\max})$ if he has not done so previously. Therefore, by Lemma 25, the leader receives, for every $p_g$, a kernel for $p_g$ of certificates for each state $(v_g^{\max}, ts_g^{\max})$ after $2\Delta$ time steps. More formally, Lemma 25 establishes that for every guild member $p'_g \in Q_\ell$ and every maximal state $(v_g^{\max}, ts_g^{\max})$ of some guild member $p_g \in Q_\ell$, the singleton subset $K := \{(v_g^{\max}, ts_g^{\max}, w) \in \mathit{witnesses}\}$ satisfies $\mathit{cert}_{g'}(K, v_g^{\max}, ts_g^{\max})$.

This shows that the leader satisfies $C_\ell(Q_\ell, \mathit{received}, \mathit{witnesses}, v^{\max})$ (line 193), where $(v^{\max}, ts^{\max})$ is the state with the maximal timestamp among the states $(v_g^{\max}, ts_g^{\max})$ for the processes $p_g$ in the leader's guild quorum $Q_\ell$. To see this, recall that as stated above, for every process in $p_g \in Q_\ell$, the leader satisfies either $\mathit{unbound}_g$ or $\mathit{couldbind}_g$. In the first case $V_g$ is satisfied. In the second case, if $ts_g^{\max} = ts^{\max}$, then again by Lemma 22, it follows that $v^{\max} = v_g^{\max}$, and hence $\mathit{bind}_g(S^g, W_g, v^{\max}, ts^{\max})$ is satisfied, where $W_g \subseteq \mathit{witnesses}$ contains the kernel for $p_g$ of witnesses for the state $(v^{\max}, ts^{\max})$. Thus, $V_g$ is satisfied. Recall that the existence of $W_g$ has been established above by showing that *witnesses* contains a kernel for $p_g$ of witnesses for every state $v_g^{\max}, ts_g^{\max}$ and every guild member $p_g$.

In the scenario where $ts_g^{\max} < ts^{\max}$, then $\mathit{debind}_g(S^g, W_g, W'_g, v^{\max}, ts^{\max})$ is satisfied, where $W_g \subseteq \mathit{witnesses}$ contains the kernel for $p_g$ of witnesses that certify $(v_g^{\max}, ts_g^{\max})$ and $W'_g \subseteq \mathit{witnesses}$ contains the kernel for $p_g$ of witnesses that certify $(v^{\max}, ts^{\max})$.
Hence every process $p_g \in Q_\ell$ satisfies $V_g$ for value $v^{\max}$, and therefore $p_\ell$ satisfies $C_\ell(Q_\ell, \mathit{received}, \mathit{witnesses})$ at most after $3\Delta$ time steps after the last guild member started the epoch. □

We now show that APBFT implements the Asymmetric Epoch Consensus interface.

**Theorem 27.** *The APBFT algorithm implements the Asymmetric Epoch Consensus interface*

*Proof.* We first show *Weak Validity*. First, Observe that all processes are guild members, since all processes are correct. A process decides $v$ in epoch *ets* only after receiving PRECOMMIT messages for $v$ from a quorum for itself (line 145). Furthermore, a process only sends a PRECOMMIT message for $v$ after receiving WRITE messages for $v$ from a quorum for itself in epoch *ets* (line 219). Let $p_g$ be the first process to send a WRITE message for $v$ in some epoch *ets'*. By the minimality of $p_g$, this can only occur after the process has returned $v$ from the first phase of the algorithm (line 204). The leader $\ell'$ of epoch *ets'* sends $v$ to $p_g$ (line 198) only after satisfying the $C_{\ell'}$ predicate (line 193), which requires collecting states and certificates such that a quorum $Q_{\ell'}$ for the leader satisfies the *bind*, *debind*, or *unbound* predicate for value $v$. Observe that $\mathit{cert}_j(\cdot, v, \cdot)$ is unsatisfiable for all processes $p_j \in Q_{\ell'}$, since otherwise, by Lemma 20, this would contradict the minimality of $p_g$. This implies, by definition, that for all processes $p_j \in Q_{\ell'}$, both $\mathit{bind}_j(\cdot, \cdot, v, \cdot)$ and $\mathit{debind}_j(\cdot, \cdot, \cdot, v, \cdot)$ are unsatisfiable. Therefore, all processes in $Q_{\ell'}$ must satisfy the *unbound* predicate for $C_{\ell'}(Q_{\ell'}, \mathit{states}, \mathit{witnesses}, v)$ to be satisfied. Thus, the leader satisfies the condition on line 194, and therefore $v$ is the proposal of $\ell'$, establishing *Weak Validity*.

The *Agreement* property follows directly from Lemma 22, since correct processes only send PRECOMMIT messages for values that they have written to their local state in a given epoch (line 140).

*Integrity* is ensured by the $\neg$*decided* guard on the event that triggers *Decide* (line 145). The guard is set to TRUE before deciding, and is only reset once a new epoch starts (line 168).

To show *Lock-In:* Assume a process $p_i$ has decided on value $v$ in some epoch *ets*. For the sake of contradiction, assume that a process $p_j$ decides on a value $v' \neq v$ in some epoch *ets'* $>$ *ets*. It follows that $p_j$ has received a

PRECOMMIT message for $v'$ from a quorum $Q$ for itself (line 145). $Q$ contains a guild member $p_k$, who has received a quorum for itself of WRITE messages for $v'$ (line 219), which itself includes a guild member. Let $p_h$ be the first guild member to write $v$ in epoch *ets*′. By the minimality of $p_h$ and Lemma 7, $p_h$ cannot have written $v'$ after receiving a kernel for itself of WRITE messages for $v'$ (line 215), and must therefore have returned $v'$ from the first phase of the algorithm (line 204). This implies that $p_h$ satisfied predicate $V_k(\cdot, \cdot, \cdot, v')$ (line 203). Lemma 24 then directly contradicts the assumption that $p_i$ decided $v$ in epoch *ets* $<$ *ets*′.

To show *Termination*, recall that Lemma 26 establishes that a quorum $Q \in \mathcal{Q}_\ell$ for $\ell$ returns a decision value $v$ from the first phase of the algorithm after at most $GST + 3\Delta$ time steps. After returning from the first phase (line 204), these processes write $v$, sending a WRITE message to all peers (line 205). By Lemma 6, this implies that every wise process receives a WRITE message for $v$ from a kernel for itself. Hence, after at most $GST + 4\Delta$ time steps, every wise process satisfies the kernel condition on line 215 and also broadcasts a WRITE message for $v$ to its peers. Therefore, by the availability condition, every wise process receives WRITE messages for $v$ from a quorum for itself after at most $GST + 5\Delta$ time steps. Hence, every wise process sends a PRECOMMIT message for $v$ (line 219), and subsequently receives PRECOMMIT messages from a quorum for itself after at most $GST + 6\Delta$ time steps, and decides $v$ (line 145).

Finally, to show *One for All*, if a wise process decides value $v$ in epoch *ets*, it must have received PRECOMMIT messages from a quorum $Q$ for itself line 145). $Q$ must contain at least one wise process, which only sends a PRECOMMIT message for $v$ after receiving WRITE messages for $v$ from a quorum $Q'$ for itself line 219). By Lemma 6, $Q'$ contains a kernel for every wise process. Hence, every wise process must eventually receive a kernel for itself of WRITE messages for $v$, and therefore must eventually write $v$ itself line 215). As argued for the *Termination* condition, this ensures that every wise process eventually receives a quorum for itself of WRITE messages for $v$, and hence eventually receives a quorum for itself of PRECOMMIT messages for $v$, and thus eventually decides $v$. □

## 6.3 Leader-Driven Consensus Algorithm

In this final section, we present the glue code that combines the Asymmetric Epoch Change and Asymmetric Epoch Consensus abstractions into a single consensus algorithm implementing Asymmetric Weak Byzantine Consensus.

The Leader-Driven Consensus Algorithm is shown in Algorithm 13. Upon receiving a *aec.StartEpoch* event from the epoch change abstraction, the algorithm relays this information to the epoch consensus abstraction by triggering its corresponding *aep.StartEpoch* event. Additionally, a timeout is started. The timeout duration is set to a constant multiple of the epoch number, ensuring that the process increases its wait time with each successive epoch.

If the timer expires, the process suspects the leader to be faulty and issues a *aec.Complain* event to the epoch change abstraction. When a value is proposed, it is directly forwarded to the epoch consensus instance. Upon receiving a *aep.Decide* event from the epoch consensus instance, the timeout is canceled. If the process has not yet made a decision, it decides on the received value.

**Theorem 28.** *The Leader-Driven Consensus Algorithm implements Weak Consensus.*

*Proof.* The *Weak Validity* property follows directly from the *Weak Validity* property of the Asymmetric Epoch Consensus abstraction. Assume all processes are correct, and that process $p_i$ decides on a decision value $v$. Process $p_i$ decides only after an *aep.Decide*$(v)$ event occurs (line 229). The *Weak Validity* property of Asymmetric Epoch Consensus ensures that $v$ was *aep.Proposed* by some correct process $p_j \in \mathcal{P}$. However $p_j$ only *aep.Proposes* $v$ immediately after $v$ has been *awc.Propsed* by $p_i$ (line 223), thereby establishing the validity property.

Concerning *Integrity*, observe that before deciding, a process sets its *decided* guard to TRUE (line 232) and only decides when ¬*decided* holds (line 231). Since *decided* is only set to FALSE upon initialization (line 221) *Integrity* is ensured.

To show *Agreement*, assume that some wise process $p_i$ decides $v$, and another wise process $p_j$ decides $v'$. Both processes can only decide after an *aep.Decide* event occurs (line 229). Hence, the *Agreement* and *Lock-In* properties of Asymmetric Epoch Consensus ensure that $v = v'$. The *Agreement* property applies when both processes decide in the same epoch of the epoch consensus instance, and the *Lock-In* applies when they decide in different epochs.

Finally we show the *Termination* property. For the sake of contradiction, assume that some guild member $p_g \in \mathcal{G}_{\text{max}}$ never decides. There are two cases to consider. Either $p_g$ starts an infinite sequence of epochs, or $p_g$ remains in some epoch indefinitely. In the first case, by the *Eventual Entry* property, $p_g$ can only remain indefinitely in an epoch *ets* if all guild members remain in the same epoch indefinitely. Therefore, the *Eventual Progression* property of Asymmetric Epoch Change implies that no kernel for all processes ever complains in epoch *ets*. Since, by Lemma 8, the maximal guild $\mathcal{G}_{\text{max}}$ contains a kernel for all processes, it follows that some guild member $p_k$ never complains in epoch *ets*. However, a correct process always complains in an epoch it has started unless it decides in

**Algorithm 13** Leader-Driven Consensus Algorithm (Process $p_i$).

```
    implements
        AsymmetricWeakByzantineConsensus, instance awc
    uses
        AsymmetricEpochChange, instance aec
        AsymmetricEpochConsensus, instance aep
    state
221:    decided ← FALSE
222:    (ets, ℓ) ← (0, ⊥)

223: upon event awc.Propose(v) do
224:    aep.Propose(v)

225: upon event aec.StartEpoch(newts, newℓ) such that newts > ets do
226:    (ets, ℓ) ← (newts, newℓ)
227:    trigger aep.StartEpoch(newts, newℓ)
228:    StartTimer((newts + 1) · Δ)

229: upon event aep.Decide(v) do
230:    CancelTimer()
231:    if ¬decided then
232:        decided ← TRUE;
233:        trigger awc.Decide(v);

234: upon event Timeout do
235:    trigger aec.Complain(ets)
```

that epoch. Therefore, by the *One for All* property of Asymmetric Epoch Consensus, every guild member decides in epoch *ets*. In particular $p_g$ decides in epoch *ets*, a contradiction.

Now consider the latter case, where $p_g$ starts an infinite number of epochs. By the *Eventual Entry* property, this implies that every guild member starts an infinite number of epochs. Since the guild members complain in increasing time intervals, the precondition of the *Unbounded Leadership* property of Asymmetric Epoch Change is satisfied. Combined with the *Uniform Leadership* property, this implies that for any duration $d$ there is some epoch $ets_d$, such that all guild members are simultaneously in epoch $ets_d$ for at least that duration and the leader of $ets_d$ is a guild member. Hence it follows that the guild members are simultaneously in an epoch *ets* that satisfies the precondition of the *Termination* property of the epoch consensus abstraction. Therefore, all guild members decide in epoch *ets*, a contradiction.

□

# 7
# Experimental Evaluation

In this chapter we discuss the experimental evaluation of the PAPBFT algorithm [23] and the Asymmetric Randomized Consensus algorithm [19].

**Benchmarking Parameters.** We empirically investigate the scalability and impact of faulty processes on the running time of the two algorithms. All experiments are performed on three quorum systems consisting of 5, 6, and 7 processes. These quorum systems are taken directly from the example systems provided in the paper by Alpos et al. [1], illustrated in Examples 1, 2, and 4. We restrict our analysis to this small collection of quorum systems, since constructing asymmetric quorum systems is a non-trivial task with no known general procedure. Moreover, the benchmark is not suitable for large quorum systems.

To assess scalability, we measure the performance of the algorithms as the number of processes in the quorum system increases.

To evaluate the impact of faulty processes, we compare the algorithms' performance under failure-free conditions to their performance when the maximum allowable number of processes fail. The maximum number of failures is determined by determining the smallest guild in the system while allowing all other processes to fail. Failed nodes are modeled as non-responsive processes that do not reply to messages from their peers. Byzantine nodes are simulated solely as crash faults, since modeling more complex malicious behavior would complicate the benchmarking process without any anticipated impact on the performance of the algorithms compared to crash faults. Verifying this hypothesis is left as future work.

**Benchmark Metric.** The performance of the algorithm is measured using the **quorum response time** of the quorum system. To perform this measurement, a dedicated process referred to as the client is introduced. The client is responsible for initiating and observing the execution of the consensus algorithm. The measurement begins when the client sends a start message to all processes in the quorum system that participate in the consensus algorithm, along with a value for them to propose.

Each replica executes the consensus protocol independently. When a replica completes the protocol and reaches a decision, it sends a decide message with the decided value back to the client. The client collects these responses and terminates the measurement once it has received decision messages from any quorum in the asymmetric quorum system.

More formally, for a given quorum system, the client stops the measurement once:

$$\exists p_i \in \mathcal{P}, Q \in \mathcal{Q}_i : \text{the client has received a decide message from all processes in } Q.$$

This duration is referred to as the **quorum response time** of the consensus algorithm. The measuring procedure is illustrated in Figure 7.1.

**Hardware Setup.** All experiments were conducted on a single computer with an AMD Ryzen 7 5800X processor. The network was implemented by communicating locally using the loopback interface (i.e., 127.0.0.1).

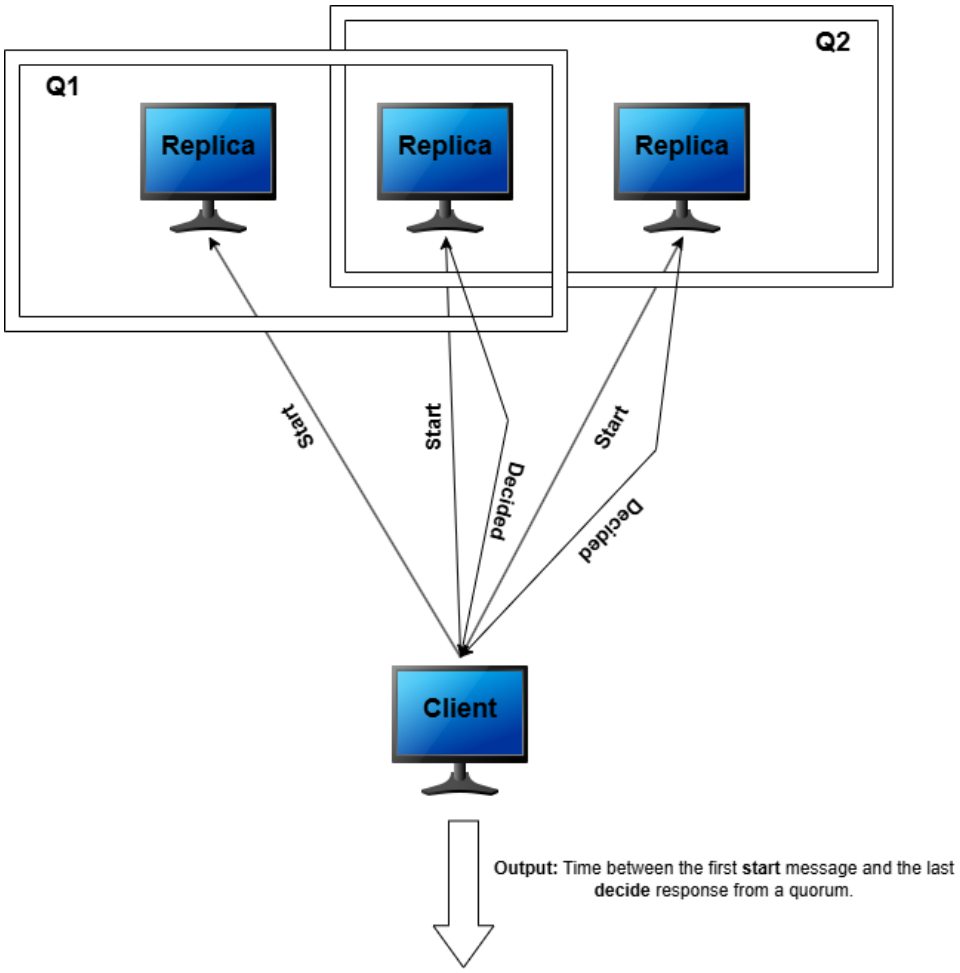



**Figure 7.1.** Client/Replicas architecture for measuring the **quorum response time**. The client starts the measurement by sending a start message to all the replicas. The replicas run the consensus algorithm and respond back to the client once they have decided. As soon as the client is informed of some quorum deciding, the measurement is terminated. In the illustration quorum $Q_2$ has decided

**Repetition of Measurements.** To increase the stability of the measurements, each experiment was repeated 50 times, and the results report the average metric. Furthermore, in each repetition, the identifiers of the processes in the asymmetric quorum system were shuffled to average out the influence of process labeling on the measurement. This labeling has a particularly strong effect on the leader-driven PAPBFT consensus algorithm, where it determines the sequence in which processes are elected as leaders of the epoch consensus instances.

We settled on this number of repetitions because, given the observed standard deviation of the measurements, we believe it is sufficient to obtain a reliable estimate of the true underlying mean.

**Asymmetric Common Coin Implementation.** As mentioned in Chapter 5, the common coin of the Asymmetric Randomized Consensus algorithm has not been implemented. To make the algorithm operational, the common coin was substituted with a local random bit generator. To ensure the *Matching* property of the Asymmetric Common Coin abstraction (Module 2), each random bit generator was initialized with the same seed. This implementation, however, violates the *Unpredictability* property of the abstraction. Nevertheless, this violation does not affect the correctness of the algorithm in the experiment, since the emulated Byzantine processes only crash and therefore do not exploit it.

**Scalability and Performance Comparison.** Figures 7.2a and 7.2b show the response times of the two algorithms as the number of processes in the asymmetric quorum system increases and their structures change. Figure 7.2a presents results for executions in which all processes are correct, while Figure 7.2b shows the scenario where the maximal number of processes fail. A direct comparison of the execution times under both conditions is further summarized in Table 7.1.

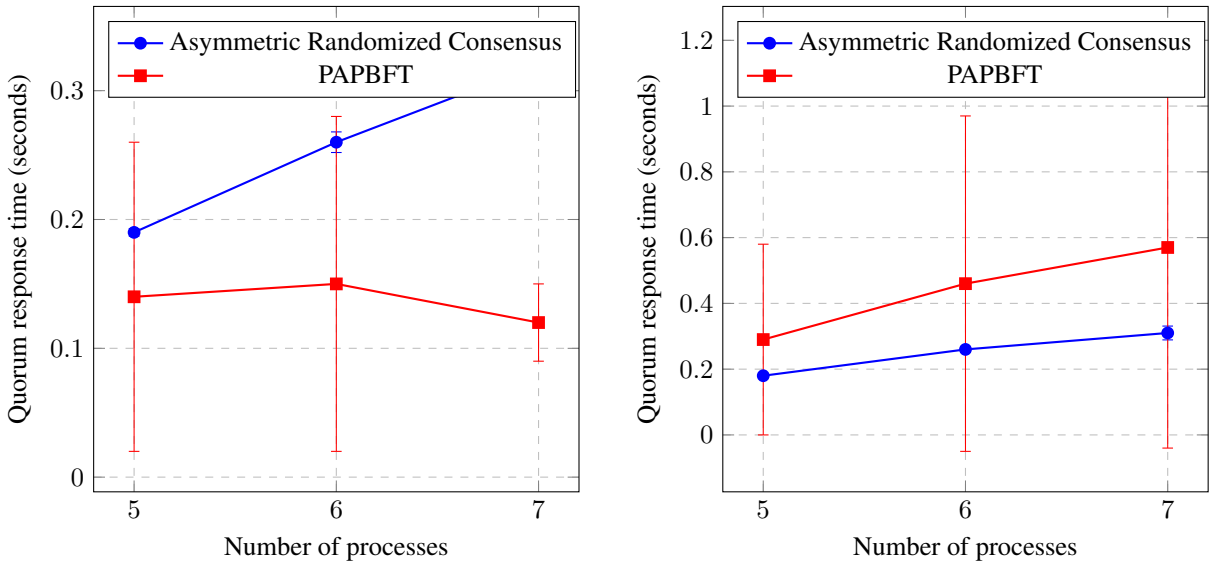
Figure 7.2a demonstrates that when all processes are correct, PAPBFT exhibits good scalability. There is no clear upward trend in the quorum response time as the system size increases. In contrast, Asymmetric Randomized Consensus shows poor scalability: as the number of processes increases, the response time increases steadily. The algorithm requires on average 1.75 times longer on 7 than on 5 processes. Moreover, in this failure-free scenario, PAPBFT clearly outperforms Asymmetric Randomized Consensus, with the margin steadily increasing as the number of processes increases. On 5 processes, PAPBFT runs on average 1.35 times faster than Asymmetric Randomized Consensus. On 7 processes, the performance gap significantly increases, with PAPBFT executing 2.66 times faster than Asymmetric Randomized Consensuss on average.

However, this picture changes dramatically in the presence of faults, as illustrated in Figure 7.2b. In this scenario, the Asymmetric Randomized Consensus algorithm significantly outperforms PAPBFT, executing between 1.5 and 1.9 times faster than PAPBFT on average.

Table 7.1 offers insight into this performance shift. The performance of the Asymmetric Randomized Consensus algorithm is barely affected by the presence of faulty processes; it remains nearly constant across both conditions. In contrast, the performance of the PAPBFT algorithm degrades substantially when faulty processes are introduced. This behavior is a direct consequence of the Leader-Driven Consensus Paradigm: agreement can only be reached during an epoch led by a correct leader. If the leaders of the first $k$ epochs are faulty, then, by design, the system spends at least $\Delta \cdot (1 + 2 + ... + k) = \Delta \cdot (\frac{k \cdot (k+1)}{2})$ time steps making no progress. Consequently, the choice of $\Delta$ and the number of faulty processes, both in absolute terms and relative to the total number of processes, have a significant impact on performance. In our experiments, $\Delta$ was set to 200 ms. This effect is clearly visible in the increasing response times of PAPBFT in Figure 7.2b. The relative number of failures are $\frac{2}{5}$, $\frac{3}{6}$, and $\frac{4}{7}$ for the three quorum systems, respectively. As the number of processes increases, both the absolute number of faulty processes and the failure rate rise, which in turn leads to the observed worsening performance of PAPBFT under faulty conditions.

The same factors also explain the higher variability of PAPBFT's response times in executions with faults compared to those without. Because leader identifiers are shuffled between repetitions, different runs encounter varying arrangements of faulty leaders, leading to differences in execution time. Moreover, as the total number of processes grows, so does the number of faulty processes and, consequently, the maximum possible length of consecutive faulty leaders. This increases the potential variation in execution times across repetitions, resulting in a higher standard deviation as the number of processes increases.

Notably, the plots show that the performance of Asymmetric Randomized Consensus is much more stable compared to PAPBFT: the standard deviation of PAPBFT's response times reaches up to 0.133 s under no faults, while Asymmetric Randomized Consensus maintains a low standard deviation of around 0.008 s.




**(a)** No faulty processes.

**(b)** Maximal faulty processes.

**Figure 7.2.** Average quorum response time of PAPBFT and Asymmetric Randomized Consensus as the number of processes increases. Error bars indicate the standard deviation of the measurement.

**Table 7.1.** Quorum response time (seconds) under different failure modes for PAPBFT and Asymmetric Randomized Consensus

| **Asymmetric quorum system** | **PAPBFT** | | **Asymmetric Randomized Consensus** | |
|---|---|---|---|---|
| | **No Failures** | **Maximal Failures** | **No Failures** | **Maximal Failures** |
| 5 Peer System | 0.14 | 0.27 | 0.19 | 0.18 |
| 6 Peer System | 0.15 | 0.48 | 0.26 | 0.26 |
| 7 Peer System | 0.12 | 0.59 | 0.32 | 0.31 |

# 8
# Future Work

We believe that the guarantees provided by the APBFT algorithm can be strengthened through a more thorough analysis of the epoch consensus primitive and a simple extension of the leader-driven consensus algorithm. More precisely, we believe that the *Agreement* and *Lock-In* properties of epoch consensus hold for all correct processes in executions with a guild. This should be derivable from Lemma 9, which can be applied in a straightforward manner to the provided proofs to establish these additional *Agreement* and *Lock-In* properties. This would allow for a strengthening of the *Termination* condition of the APBFT algorithm, guaranteeing termination for all wise processes when a guild exists.

Regarding the empirical evaluation of the consensus algorithms, it is important to extend the benchmark to include substantially more quorum systems with a larger variation in the number of processes, in order to increase the reliability of the results. This requires a mechanism that allows for the automatic generation of asymmetric quorum systems for different numbers of processes. To support this, the benchmarking code needs to be adapted.

In its current form, the benchmark evaluates the performance of the algorithms under faulty executions by computing the smallest guild of the quorum system and letting all other processes fail. As we have shown, this approach does not scale to quorum systems with many processes. This limitation could be addressed by instead randomly sampling, at each repetition of an experiment, some minimal guild of the quorum system, which can be done efficiently.

It would also be of interest to extend the benchmark with additional consensus algorithms. One candidate of particular interest is the DAG-based consensus algorithm proposed by Amores-Sesar et al. [3].

# 9 Conclusion

In this work, we presented the first empirical evaluation of two consensus paradigms tailored for the asymmetric trust setting: the leader-driven protocol PAPBFT and Asymmetric Randomized Consensus a protocol designed for fully asynchronous environments. Our evaluation demonstrates that PAPBFT scales more efficiently than Asymmetric Randomized Consensus in fault-free scenarios, achieving significantly lower response times as the number of processes increases. However, PAPBFT's performance deteriorates substantially in the presence of faulty processes due to its reliance on a correct leader for progress, causing delays when faulty leaders are repeatedly selected. In contrast, Asymmetric Randomized Consensus exhibits stable performance regardless of faults, maintaining consistent response times with minimal variability.

Alongside these experiments, we examined the algorithmic underpinnings required to make asymmetric trust consensus practical. We proved that determining the tolerated system of an asymmetric quorum system is NP-hard, and introduced the superset recognizer abstraction as a practical way to work around this barrier. Crucially, an efficient superset recognizer for each quorum system in an asymmetric quorum system can be leveraged to build superset recognizers for both its kernel system and its tolerated system. This means our consensus algorithms can be implemented entirely on top of quorum superset recognizers, avoiding the need to explicitly enumerate complex system structures while retaining efficiency.

Additionally, we proposed a novel variant of the PBFT algorithm that does not depend on the tolerated system of the quorum system. Our approach introduces new techniques for managing unequal role distributions among processes in asymmetric trust environments. Finally, we provide a complete correctness proof of the PBFT algorithm in the asymmetric trust setting by fully specifying the heartbeat mechanism, ensuring its reliability and robustness.

# Bibliography


[1] Orestis Alpos, Christian Cachin, Björn Tackmann, and Luca Zanolini. Asymmetric distributed trust. *Distributed Comput.*, 37(3):247–277, 2024. URL: https://doi.org/10.1007/s00446-024-00469-1.

[2] Ignacio Amores-Sesar, Christian Cachin, and Jovana Micic. Security analysis of ripple consensus. *CoRR*, abs/2011.14816, 2020. URL: https://arxiv.org/abs/2011.14816.

[3] Ignacio Amores-Sesar, Christian Cachin, Juan Villacis, and Luca Zanolini. Dag-based consensus with asymmetric trust [extended version]. *CoRR*, abs/2505.17891, 2025. URL: https://doi.org/10.48550/arXiv.2505.17891.

[4] Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. Liveness and latency of byzantine state-machine replication. *Distributed Comput.*, 37(2):177–205, 2024. URL: https://doi.org/10.1007/s00446-024-00466-4.

[5] Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *CoRR*, abs/1807.04938, 2018. URL: http://arxiv.org/abs/1807.04938.

[6] Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011. URL: https://doi.org/10.1007/978-3-642-15260-3.

[7] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *OSDI*, pages 173–186. USENIX Association, 1999. URL: https://dl.acm.org/citation.cfm?id=296824.

[8] Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In *ASIACRYPT*, volume 4833 of *Lecture Notes in Computer Science*, pages 357–375. Springer, 2007. URL: https://doi.org/10.1007/978-3-540-76900-2_22.

[9] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. URL: https://doi.org/10.1145/42282.42283.

[10] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985. URL: https://doi.org/10.1145/3149.214121.

[11] Álvaro García-Pérez and Alexey Gotsman. Federated byzantine quorum systems (extended version). *CoRR*, abs/1811.03642, 2018. URL: http://arxiv.org/abs/1811.03642.

[12] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In *PODC*, pages 165–175. ACM, 2021. URL: https://doi.org/10.1145/3465084.3467905.

[13] Lukasz Lachowski. Complexity of the quorum intersection property of the federated byzantine agreement system. *CoRR*, abs/1902.06493, 2019. URL: http://arxiv.org/abs/1902.06493.

[14] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. URL: https://doi.org/10.1145/279227.279229.

[15] Xiao Li, Eric Chan, and Mohsen Lesani. Quorum subsumption for heterogeneous quorum systems. In *DISC*, volume 281 of *LIPIcs*, pages 28:1–28:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. URL: https://doi.org/10.4230/LIPIcs.DISC.2023.28.

[16] Ethan MacBrough. Cobalt: BFT governance in open networks. *CoRR*, abs/1802.07240, 2018. URL: http://arxiv.org/abs/1802.07240.

[17] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Comput.*, 11(4):203–213, 1998. URL: https://doi.org/10.1007/s004460050050.

[18] David Mazières, Nicolas Barry, Giuliano Losa, Jed McCaleb, and Stanislas Polu. The stellar consensus protocol: A federated model for internet-level consensus, 2015. URL: https://stellar.org/papers/stellar-consensus-protocol.pdf.

[19] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous binary byzantine consensus with t $<$ n/3, o(n2) messages, and O(1) expected time. *J. ACM*, 62(4):31:1–31:21, 2015. URL: https://doi.org/10.1145/2785953.

[20] David Schwartz, Noah Youngs, and Arthur Britto. The ripple protocol consensus algorithm. Technical report, Ripple Labs Inc., 2014. URL: https://ripple.com/files/ripple_consensus_whitepaper.pdf.

[21] Isaac C. Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C. Myers. Heterogeneous paxos: Technical report. *CoRR*, abs/2011.08253, 2020. URL: https://arxiv.org/abs/2011.08253.

[22] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan-Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *PODC*, pages 347–356. ACM, 2019. URL: https://doi.org/10.1145/3293611.3331591.

[23] Luca Zanolini. *Asymmetric Trust in Distributed Systems*. PhD thesis, University of Bern, Faculty of Science, Bern, Switzerland, 2023. URL: https://doi.org/10.48549/4481.