Asymmetric Trust in Distributed Systems

Inaugural Dissertation

of the Faculty of Science,
University of Bern

Presented by

Luca Zanolini

from Brescia, Italy

Supervisor of the Doctoral Thesis:

Prof. Dr. Christian Cachin
University of Bern, Switzerland

Asymmetric Trust in Distributed Systems

Inaugural Dissertation

of the Faculty of Science,
University of Bern

Presented by

Luca Zanolini

from Brescia, Italy

Supervisor of the Doctoral Thesis:

Prof. Dr. Christian Cachin
University of Bern, Switzerland

Accepted by the Faculty of Science.

Bern, 6th July 2023

The Dean
Prof. Dr. Marco Herwegh

Acknowledgments

First of all, I wish to express my gratitude to my advisor, Prof. Dr. Christian Cachin. He has been more than just an advisor, demonstrating through his actions what it truly means to be both a mentor and a leader. His faith in my abilities, his patience, and his constant encouragement have been fundamental in shaping the researcher I am today. To Prof. Dr. Christian Cachin, I owe a debt of gratitude that goes beyond words.

To my girlfriend, Stefanie, I extend my deepest thanks. Her support, her love, her patience during the stressful times, and her belief in me have been my guiding lights through the stormy seas of this journey.

To my parents, whose love has been the foundation on which I built my dreams, thank you. Your sacrifices and your constant encouragement have been my beacon, guiding me through every challenge.

This journey would have been entirely different without my dearest friends - Matteo, Giordano, and Gianluca. Their support, friendship, and shared growth have been my pillars of strength throughout this path.

To my colleagues who have journeyed alongside me in this academic endeavor, I extend my sincere appreciation. Your intellectual contributions have been integral to my development and success in this path. I am particularly indebted to Duc, Jovana, and Orestis, whose support and insights have been invaluable.

Last, but not least, I wish to express my gratitude to my therapist, Lori Hughes. Lori's expertise, compassion, and guidance have been crucial in aiding me to cope with my challenges, allowing me to find strength within myself I didn't know existed.

Abstract

Secure distributed systems rely on trust. A trust assumption specifies the failures that a system can tolerate and determines the conditions under which it operates correctly. Trust in secure distributed systems has traditionally been devised in a symmetric way, where every participant in the system adheres to the same, global notion of trust through a fail-prone system, a collection of subsets of participants that can fail together and are tolerated to fail in an execution. However, recent advances in the field, particularly in the context of blockchain networks, have led to the development of new models of trust that allow participants to express their own beliefs in a subjective way.

In this thesis, we study in detail and extend with new results the model of asymmetric trust first introduced by Damgård, Desmedt, Fitzi, and Nielsen (ASIACRYPT 2007), and carried on by Cachin and Tackmann (OPODIS 2019). We develop the first asynchronous consensus protocol that works in this model and evaluate its properties and resilience against Byzantine failures. This protocol results in an optimal randomized Byzantine consensus protocol with subjective, asymmetric trust and constant expected running time.

Furthermore, we define a composition rule that allows for the joining of distributed systems based on asymmetric trust. These composition rule ensures that if the original systems allow for running a particular protocol (guaranteeing consistency and availability), then the joint system will as well, while tolerating as many faults as possible.

Finally, we expand the asymmetric trust model to work in permissionless settings, where the number of participants in the system is not known, such as in blockchain systems. We do so by enabling processes not only to make assumptions about failures, but also to make assumptions about the assumptions of other processes. The resulting model generalizes existing models such as the classic fail-prone system model

[Malkhi and Reiter, 1998] and the asymmetric fail-prone system model.

Our contributions provide new insights into the notion of trust in secure distributed systems and offer solutions for enhancing the security and reliability of these systems in the presence of Byzantine failures. The results of this thesis have the potential to improve the security of distributed systems, particularly in applications such as blockchain systems.

Contents

Chapter 1

Introduction

Secure distributed systems rely on *trust*. Trust specifies the failures that a system can tolerate and determines the conditions under which it may operate correctly. Implicitly, this determines the trust in certain components to be correct. Traditionally, trust has been expressed globally, through an assumption on the number or kind of faulty participants (or, processes), which is *shared by every process*. An example of this is the well-known threshold fault assumption: the system tolerates up to a finite and limited number of faulty processes in the system; no guarantees can be given beyond this about the correct execution of protocols.

In fault-tolerant replicated systems, trust is defined through the notion of *fail-prone system* [26], a collection of subsets of processes, called *fail-prone sets*, such that each of them contains all the processes that may at most fail together, and that are tolerated to fail, during a protocol execution. Fail-prone systems are useful tools for the design of distributed protocols due to their relationship to quorum systems or, more specifically, to *Byzantine* quorum systems.

Byzantine quorum systems have been formalized by Malkhi and Reiter [26] to express trust assumptions operationally, under Byzantine failures, i.e., where faulty processes may behave arbitrarily. A Byzantine quorum system is a collection of subsets of processes, called *quorums*, with the properties that each pair of quorums intersect in at least a correct process (Consistency), and at least a quorum made exclusively by correct processes must exist (Availability).

Many distributed algorithms (implementing, e.g., Byzantine reliable broadcast or consensus) are parameterized by a quorum system $\mathcal{Q}$ and

their guarantees hold under the assumptions of a fail-prone system $\mathcal{F}$ if and only if Consistency and Availability of $\mathcal{Q}$ hold. This allows the designers of a distributed system to make assumptions about failures, pick a corresponding quorum system, and then choose among existing algorithms to solve the desired synchronization problem.

Motivated by the requirements of more flexible trust models, particularly in the context of blockchain networks, new approaches to trust have been explored. It is evident that a common trust model cannot be imposed in an open and decentralized or permissionless environment. Instead, every participant in the system should be free to choose who to trust and who not to trust. Damgård, Desmedt, Fitzi, and Nielsen [15], and Cachin and Tackmann [9], extend Byzantine quorum systems to permit subjective trust by introducing *asymmetric Byzantine quorum systems*. They let every process specify their own fail-prone system and quorum system, and global system guarantees can be derived from these personal assumptions. How can we formally model asymmetric trust in distributed protocols? Is it possible to generalize standard distributed protocols to work with this more flexible trust? If so, what guarantees can be obtained from them, and to whom are they addressed?

In the first part of this work, we give an answer to these questions. In particular, in Chapter 4 we recall the theory behind both asymmetric trust, as originally introduced [15], and asymmetric Byzantine quorum systems [9], we present fundamental results on two new type of correct processes, namely *wise* and *naïve* processes. We observe that one can reliably guarantee properties of protocols only to the former ones, i.e., those who make the "right" trust assumption in a protocol execution. In particular, we show that, under certain conditions, guarantees of distributed protocols can only be given for a subset of the wise processes that form a so-called *guild*.

Moreover, we study the problem of *composing* asymmetric trust assumptions. In particular, starting from two or more running distributed systems, each one with its own subjective assumption, how can they be combined, so that their participant processes are joined and operate together? We formulate the problem of composing asymmetric Byzantine quorum systems and give methods for assembling trust assumptions from different, possibly disjoint, systems to a common model. We do so by introducing composition rules for trust assumptions in the asymmetric-trust model. Our methods describe the resulting fail-prone systems and the corresponding Byzantine quorum systems.

In Chapter 5 we start devising protocols, and we present the first

asynchronous Byzantine consensus protocol with asymmetric trust. It uses randomization, provided by an asymmetric common-coin protocol, to circumvent the impossibility of (purely) asynchronous consensus. Our protocol takes up the randomized and signature-free implementation of consensus by Mostéfaoui, Moumen, and Raynal [29], [30]. The protocol of Mostéfaoui, Moumen, and Raynal comes in multiple versions. The original one, published at PODC 2014 [30], suffers from a subtle and little-known liveness problem [36]: an adversary can prevent progress among the correct processes by controlling the messages between them and by sending them values in a specific order. The subsequent version (JACM 2015) [29] resolves this issue, but requires many more communication steps and adds considerable complexity. Our asymmetric asynchronous Byzantine consensus protocol is based on the simpler version (PODC 2014). We first revisit this and show in detail how it is possible to violate liveness. We propose a method that overcomes the problem, maintains the elegance of the protocol, and does not affect its appealing properties. Based on this insight, we show how to realize asynchronous consensus with asymmetric trust, again with a protocol that maintains the simplicity of the original approach of Mostéfaoui, Moumen, and Raynal [30]. Furthermore, we study the problem of how to implement leader-based consensus protocols with asymmetric trust. We present the problem and give a simple, possible solution in the context of PBFT [13].

One implicit requirement of the asymmetric-trust model is that the knowledge of the full system membership is required. In particular, even though participants are free to make their own failure assumptions or choose their own quorums, maintaining Consistency requires compatible assumptions (in the sense that the resulting quorums will sufficiently intersect) and thus prior synchronization among every process in the system, which is not desirable in a permissionless system. More in general, global assumptions implying the intersection of quorums are problematic in a permissionless setting because they postulate some form of pre-agreement or common knowledge, which might be hard to achieve in practice.

Interestingly, the Stellar network (https://stellar.org [28]), a deployed blockchain system based on quorums, is able to maintain safety and liveness without requiring that processes choose intersecting quorums. Instead, processes choose *quorum slices* that need not intersect, and the quorums of a process are defined in terms of the slices of other processes. Consensus can then be solved within (disjoint) subsets of processes, called a intact sets $\mathcal{I}$; an intact set is a set of correct processes

such that every two processes in $\mathcal{I}$ have all their own quorums that intersect in at least a member of $\mathcal{I}$ and such that $\mathcal{I}$ is itself a quorum for every of its members.

In Chapter 6 we observe that quorum slices can be interpreted as a new kind of failure assumptions: a process assumes that at least one of its quorum slices is made exclusively of processes that do not fail and make correct assumptions. In other words, a process's assumption are not only about failures, but also about whether other processes make correct assumptions. In practice, the Stellar model makes it easier for processes, compared to standard models, to achieve quorum intersection by relying on the failure assumptions of other processes that might have more knowledge about the system than they have. In particular, we show that this new kind of failure assumptions yield a generalization of the theory of fail-prone systems (i.e., classic, global, fail-prone systems are a special case) which allows to obtain intersecting quorums even when processes do not know any common third party. Moreover, based on this, we introduce the notion of permissionless fail-prone system from which it is possible to derive a permissionless quorum system. Our model also leads to a characterization of the Stellar model with standard formalism [9], [15], [26].

Finally, we implement a single-writer multi-reader register with permissionless quorum systems, and we adapt the Bracha broadcast [5] to work in our model, thereby offering a new toolbox for the design of permissionless distributed systems.

In Chapter 2 we discuss the related works. Prerequisites for this work are presented in Chapter 3, and conclusions are drawn in Chapter 7.

Chapter 2

Related Work

Flexible trust structures have recently received a lot of attention [9], [15], [18], [24], [25], [28], primarily motivated by consensus protocols for blockchains, as introduced by Ripple (https://ripple.com) and Stellar (https://stellar.org). According to the general idea behind these models, processes are free to express individual, *subjective* trust choices about other processes, instead of adopting a common, global view of trust.

Damgård, Desmedt, Fitzi, and Nielsen [15] define the basics of *asymmetric trust* for secure computation protocols. This model is strictly more powerful than the standard model with symmetric trust and abandons the traditional global failure assumption in the system. Moreover, they present several variations of their asymmetric-trust model and sketch synchronous protocols for broadcast, verifiable secret sharing, and general multi-party computation.

Mazières [28] introduces a new model for consensus called *federated Byzantine agreement* (FBA) and uses it to construct the *Stellar consensus protocol* [23]. In FBA, every process declares *quorum slices* – a collection of trusted sets of processes sufficient to convince the particular process of agreement. These slices are subsets of a *quorum*, which is a set of processes sufficient to reach agreement. More precisely, a quorum is defined as a set of processes that contains one slice for each member, and all quorums constitute a *federated Byzantine quorum system* (FBQS).

Byzantine quorum systems have originally been formalized by Malkhi and Reiter [26] and exist in several forms; they generalize the classical quorum systems [31] aimed at tolerating crashes to algorithms with

Byzantine failures. Moreover, Malkhi and Reiter [26] introduce the notion of fail-prone system which is then required in order to define a Byzantine quorum system.

García-Pérez and Gotsman [18] study the theoretical foundations of a FBQS, build a link between FBQS and the classical Byzantine quorum systems, and show the correctness of broadcast abstractions over federated quorum systems. Moreover, they investigate decentralized quorum constructions by means of FBQS. Finally, they propose the notion of subjective dissemination quorum system, where different participants may have different Byzantine quorum systems and where there is a system-wide intersection property. FBQS are a way towards an extension of quorum systems in a permissionless setting.

Asymmetric Byzantine quorum systems have been introduced by Cachin and Tackmann [9] and generalize Byzantine quorum systems [26] to the model with asymmetric trust. This work also explores properties of asymmetric Byzantine quorum systems and differences to the model with symmetric trust. In particular, Cachin and Tackmann [9] distinguish between different classes of correct processes, depending on whether their failure assumptions in an execution are correct. The standard properties of protocols are guaranteed only to so-called *wise* processes, i.e., those that made the "right" trust choices. Protocols with asymmetric quorums are shown for Byzantine consistent broadcast, reliable broadcast, and emulations of shared memory. In contrast to FBQS, asymmetric quorum systems appear to be a natural extension of symmetric quorum systems.

Recently, Losa, Gafni, and Mazières [24] have formulated an abstraction of the consensus mechanism in the Stellar network by introducing *Personal Byzantine quorum systems* (PBQS). In contrast to the notions of "quorums" in standard models, their definition does not require a global intersection among quorums. This may lead to several separate *consensus clusters* such that each one satisfies agreement and liveness on its own.

Another approach for designing Byzantine fault-tolerant (BFT) consensus protocols has been introduced by Malkhi, Nayak, and Ren [25], namely *Flexible BFT*. This notion guarantees higher resilience by introducing a new *alive-but-corrupt* fault type, which denotes processes that attack safety but not liveness. Malkhi, Nayak, and Ren [25] also define *flexible Byzantine quorums* that allow processes in the system to have different faults models.

Chapter 3

Prerequisites

In this chapter we recall Byzantine quorum systems as originally introduced [26] as they constitute a fundamental part of this work. We refer to them as *symmetric* Byzantine quorum systems or, when it is clear from the context, simply Byzantine quorum systems.

Quorum systems are a key mathematical abstraction in distributed fault-tolerant computing for capturing trust assumptions. Quorums help in reaching higher availability and fault-tolerance in distributed systems [37]. From a classical point of view, a quorum system is a collection of subsets of processes, called quorums, with the property that each pair of quorums have a non-empty intersection. It is a generalization of the concept of a *majority* in a democratically organized group and it is used to ensure consistency in the context of crash failures, i.e., when processes stop executing steps [6].

However, if the failing processes deviate in any conceivable way from their algorithm, the above definition is not useful. Malkhi and Reiter [26] introduced a generalization of classical quorum systems called *Byzantine quorum systems*, strengthening the definition in a way that the pair-wise intersection contains also some correct processes.

This notion is defined with respect to a *symmetric fail-prone system* $\mathcal{F} \subseteq 2^{\mathcal{P}}$, a collection of subsets of the set $\mathcal{P}$ of *processes* (or *participants*), none of which is contained in another, such that some $F \in \mathcal{F}$ with $F \subseteq \mathcal{P}$ is called a *fail-prone set* and contains all processes that may at most fail together in some execution [26].

A fail-prone system captures an assumption on the possible failure patterns that may occur. It specifies all maximal sets of faulty processes

that a protocol should tolerate in an execution; this means that a protocol designed for $\mathcal{F}$ achieves its properties as long as the set F of actually faulty processes satisfies $F \in \mathcal{F}^*$. Here and from now on, the notation $\mathcal{A}^*$ for a system $\mathcal{A} \subseteq 2^{\mathcal{P}}$, denotes the collection of all subsets of the sets in $\mathcal{A}$, that is, $\mathcal{A}^* = \{A' | A' \subseteq A, A \in \mathcal{A}\}$.

Definition 3.1 (Symmetric Byzantine quorum system [26])**.** *A* symmetric Byzantine quorum system *for $\mathcal{F}$ is a collection of sets of processes $\mathcal{Q} \subseteq 2^{\mathcal{P}}$, where each $Q \in \mathcal{Q}$ is called a* quorum*, such that the following properties hold:*

Consistency: *The intersection of any two quorums contains at least one process that is not faulty, i.e.,*

$$\forall Q_1, Q_2 \in \mathcal{Q}, \forall F \in \mathcal{F} : Q_1 \cap Q_2 \not\subseteq F.$$

Availability: *For any set of processes that may fail together, there exists a disjoint quorum in $\mathcal{Q}$, i.e.,*

$$\forall F \in \mathcal{F} : \exists\ Q \in \mathcal{Q} :\ F \cap Q = \emptyset.$$

For example, under the common threshold failure model, the quorums are all sets of at least $\lceil \frac{n+f+1}{2} \rceil$ processes, where f is the number of processes that may fail. In particular, if $n = 3f + 1$, quorums have $2f + 1$ or more processes.

The above notion is also known as a *Byzantine dissemination quorum system* [26] and allows a protocol to be designed despite arbitrary behavior of the potentially faulty processes. The notion generalizes the usual threshold failure assumption for Byzantine faults [33], which considers that any set of f processes are equally likely to fail.

We say that a set system $\mathcal{T}$ *dominates* another set system $\mathcal{S}$ if for each $S \in \mathcal{S}$ there is some $T \in \mathcal{T}$ such that $S \subseteq T$ [17]. In this sense, a quorum system for $\mathcal{F}$ is *minimal* whenever it does not dominate any other quorum system for $\mathcal{F}$. A *maximal* set system is defined analogously.

Similarly to the threshold case, where $n > 3f$ processes overall are needed to tolerate f faulty ones in many Byzantine protocols, symmetric Byzantine quorum systems can only exist if not "too many" processes fail.

Definition 3.2 (Q^3-condition [20], [26])**.** *A fail-prone system $\mathcal{F}$ satisfies the* Q^3-condition*, abbreviated as $Q^3(\mathcal{F})$, whenever it holds*

$$\forall F_1, F_2, F_3 \in \mathcal{F} : \mathcal{P} \not\subseteq F_1 \cup F_2 \cup F_3.$$

In other words, $Q^3(\mathcal{F})$ means that no *three* fail-prone sets together cover the whole system of processes. A Q^k-condition can be defined like this for any $k \geq 2$ [20].

The following lemma considers the *bijective complement* of a process set $\mathcal{S} \subseteq 2^{\mathcal{P}}$, which is defined as $\overline{\mathcal{S}} = \{\mathcal{P} \setminus S | S \in \mathcal{S}\}$, and turns $\mathcal{F}$ into a symmetric Byzantine quorum system.

Lemma 3.3 ([26, Theorem 5.4])**.** *Given a fail-prone system $\mathcal{F}$, a symmetric Byzantine quorum system for $\mathcal{F}$ exists if and only if $Q^3(\mathcal{F})$. In particular, if $Q^3(\mathcal{F})$ holds, then $\overline{\mathcal{F}}$, the bijective complement of $\mathcal{F}$, is a symmetric Byzantine quorum system.*

The quorum system $\mathcal{Q} = \overline{\mathcal{F}}$ is called the *canonical Byzantine quorum system* of $\mathcal{F}$.

Given a symmetric Byzantine quorum system $\mathcal{Q}$, we define a *symmetric kernel* K, or simply *kernel*, as a set of processes that overlaps with every quorum in $\mathcal{Q}$.

Definition 3.4 (Kernel system)**.** *A set $K \subseteq \mathcal{P}$ is a* symmetric kernel *of a symmetric Byzantine quorum system $\mathcal{Q}$ whenever it holds*

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

This can be viewed as a consistency *property.*

We also define the symmetric kernel system $\mathcal{K}$ *of $\mathcal{Q}$ to be the set of all kernels of $\mathcal{Q}$. Given this, the minimal symmetric kernel system is a kernel system for which every kernel K satisfies*

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

For example, under a threshold failure assumption where any f processes may fail, every set of $\left\lfloor \frac{n-f+1}{2} \right\rfloor$ processes is a kernel. In particular, $n = 3f + 1$ if and only if every kernel has $f + 1$ processes.

Lemma 3.5. *For every $F \in \mathcal{F}$ and for every quorum $Q \in \mathcal{Q}$ there exists a kernel $K \in \mathcal{K}$ such that $K \subseteq Q$.*

Proof. Let $\mathcal{Q}$ be a symmetric quorum system for $\mathcal{F}$ and let $F \in \mathcal{F}$. From the consistency property of a quorum system we have that for all $Q_1, Q_2 \in \mathcal{Q}$ it holds $Q_1 \cap Q_2 \not\subseteq F$. Then, the set $K = Q_1 \setminus F \subseteq Q_1$ intersects all quorums in $\mathcal{Q}$ and is a kernel of $\mathcal{Q}$. □

Chapter 4

Asymmetric Distributed Trust

In this chapter we present the *asymmetric trust model*, introduced by Damgård, Desmedt, Fitzi, and Nielsen [15]. In the context of blockchains, different models of asymmetric trust have been proposed [9], [15], [23]–[25], [28], united by a shared principle regarding the subjectivity of truth among the participants, but differentiated by fundamental properties that determine their limitations and strengths. Our model is based on asymmetric trust, originally formalized by Damgård, Desmedt, Fitzi, and Nielsen [15], and later by Cachin and Tackmann [9], in which every process is free to make its own trust assumption and to express this with a fail-prone system. This chapter is partially based on the papers "How to trust strangers: Composition of byzantine quorum systems" [3] and "Asymmetric asynchronous byzantine consensus" [10].

4.1 Definition and preliminary results

Processes. We consider a system of n *processes* $\mathcal{P} = \{p_1, \ldots, p_n\}$ that communicate with each other. The processes interact by exchanging messages over reliable point-to-point links, specified below.

Failures. A process that follows its protocol during an execution is called *correct*. On the other hand, a *faulty* process may crash or deviate arbitrarily from its specification, e.g., when *corrupted* by an adversary;

such processes are also called *Byzantine*. We consider only Byzantine faults here and assume for simplicity that the faulty processes fail right at the start of an execution.

An *asymmetric fail-prone system* $\mathbb{F} = [\mathcal{F}_1, \ldots, \mathcal{F}_n]$ consists of an array of fail-prone systems, where $\mathcal{F}_i$ denotes the trust assumption of p_i. One often assumes $p_i \notin F_i$ for practical reasons, but this is not necessary.

An *asymmetric Byzantine quorum system* (ABQS) is defined from an asymmetric fail-prone system and extends the traditional notion of symmetric Byzantine quorum systems [26], as presented in Chapter 3.

Definition 4.1 (Asymmetric Byzantine quorum system)**.** *An* asymmetric Byzantine quorum system *for* $\mathbb{F}$ *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* $i \in [1, n]$*. The set* $\mathcal{C}_i \subseteq 2^{\mathcal{P}}$ *is called the* quorum system of p_i *and any set* $Q_i \in \mathcal{Q}_i$ *is called a* quorum (set) for p_i*. It satisfies:*

Consistency: *The intersection of two quorums for any two processes contains at least one process for which either process assumes that it is not faulty, i.e.,*

$$\forall i, j \in [1, n], \forall Q_i \in \mathcal{Q}_i, \forall Q_j \in \mathcal{Q}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : Q_i \cap Q_j \not\subseteq F_{ij}.$$

Availability: *For any process* p_i *and any set of processes that may fail together according to* p_i*, there exists a disjoint quorum for* p_i *in* $\mathcal{Q}_i$*, i.e.,*

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

Recall that the consistency condition for a symmetric Byzantine quorum system requires that at least one process in the intersection of every two quorums is correct. In the asymmetric case, quorums are subjective and defined according to the quorum system for each process. The asymmetric consistency property states that in the intersection of every two subjective quorums of two processes there exists at least one process that is correct according to one of the two processes. On the other hand, the availability condition in the above definition is a direct extension of the symmetric case, since it considers the quorum system of each process separately.

The existence of asymmetric Byzantine quorum systems can be characterized with a property that generalizes the Q^3-condition for the underlying asymmetric fail-prone systems as follows.

Definition 4.2 (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*

$$\forall i, j \in [1,n], \forall F_i \in \mathcal{F}_i, \forall F_j \in \mathcal{F}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : \mathcal{P} \not\subseteq F_i \cup F_j \cup F_{ij}$$

The following result, proved by Cachin and Tackmann [9], generalizes Lemma 3.3 for asymmetric Byzantine quorum systems.

Theorem 4.3. *An asymmetric fail-prone system* $\mathbb{F}$ *satisfies* $B^3(\mathbb{F})$ *if and only if there exists an asymmetric Byzantine quorum system for* $\mathbb{F}$*.*

Asymmetric kernels. Let $\mathbb{F} = [\mathcal{F}_1, \ldots, \mathcal{F}_n]$ be an asymmetric fail-prone system.

Given an asymmetric Byzantine quorum system $\mathbb{Q}$ for $\mathbb{F}$, an *asymmetric kernel system* for $\mathbb{Q}$ is defined analogously as the array $\mathbb{K} = [\mathcal{K}_1, \ldots, \mathcal{K}_n]$ that consists of the kernel systems for all processes in $\mathcal{P}$ with respect to $\mathbb{Q}$; a set $K_i \in \mathcal{K}_i$ is called an *asymmetric kernel*, or simply *kernel*, for p_i.

Naïve and wise processes. The faults or corruptions occurring in a protocol execution with an underlying quorum system induce a set F of actually *faulty processes*. However, no process knows F and this information is only available to an observer outside the system. In traditional symmetric Byzantine quorum systems every process in the system adheres to a global fail-prone system $\mathcal{F}$ and the set F of faults or corruptions occurring in a protocol execution is in $\mathcal{F}$. Given this common trust assumption, properties of a protocol are guaranteed at each correct process, while they are not guaranteed for faulty ones. With asymmetric quorums, there is a distinction among correct processes with respect to F, namely the correct processes that consider F in their trust assumption and those who do not. Given a protocol execution, the processes are classified in three different types:

Faulty: A process $p_i \in F$ is *faulty*.

Naïve: A correct process p_i for which $F \notin {\mathcal{F}_i}^*$ is called *naïve*.

Wise: A correct process p_i for which $F \in {\mathcal{F}_i}^*$ is called *wise*.

The naïve processes are new for the asymmetric case, as all processes are wise under a symmetric trust assumption. Protocols for asymmetric quorums cannot guarantee the same properties for naïve processes as for wise ones.

Guilds. A useful notion for ensuring liveness and consistency for protocols is that of a *guild*. This is a set of wise processes that contains at least one quorum for each member; by definition this quorum consists only of wise processes.

Definition 4.4 (Guild)**.** *Given an asymmetric fail-prone system $\mathbb{F}$, an asymmetric Byzantine quorum system $\mathbb{Q}$ for $\mathbb{F}$, and a protocol execution with faulty processes F, a* guild $\mathcal{G}$ for F *satisfies two properties:*

Wisdom: *$\mathcal{G}$ is a set of wise processes:*

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

Closure: *$\mathcal{G}$ contains a quorum for each of its members:*

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

Observe that the union of two guilds is again a guild, since the union consists only of wise processes and contains again a quorum for each member. All guilds overlap, as the next result shows.

Lemma 4.5. *In any execution with a guild $\mathcal{G}$, every two guilds intersect.*

Proof. Let $\mathcal{P}$ be a set of processes, $\mathcal{G}$ be a guild and F be the set of actually faulty processes. Furthermore, suppose that there is another guild $\mathcal{G}'$, with $\mathcal{G} \cap \mathcal{G}' = \emptyset$. Let $p_i \in \mathcal{G}$ and $p_j \in \mathcal{G}'$ be two processes and consider a quorum $Q_i \subseteq \mathcal{G}$ for p_i and a quorum $Q_j \subseteq \mathcal{G}'$ for p_j. From the definition of an asymmetric Byzantine quorum system it must hold $Q_i \cap Q_j \nsubseteq F$, with $Q_i \cap Q_j \neq \emptyset$ and $F \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^*$. It follows that there exists a wise process $p_k \in Q_i \cap Q_j$ with $p_k \in \mathcal{G}$ and $p_k \in \mathcal{G}'$. Notice also that $\mathcal{G}$ and $\mathcal{G}'$ both contain a quorum for p_k. □

It follows that every execution with a guild contains a unique *maximal guild* $\mathcal{G}_{\max}$. The next lemma shows that if a guild exists, no quorum for any process contains only faulty processes.

Lemma 4.6. *Let $\mathcal{G}_{max}$ be the maximal guild for a given execution and let $\mathbb{Q}$ be the canonical asymmetric Byzantine quorum system. Then, there cannot be a quorum $Q_j \in \mathcal{Q}_j$ for any process p_j consisting only of faulty processes.*

Proof. Given an execution with F as set of faulty processes, suppose there is a guild $\mathcal{G}_{\max}$. This means that for every process $p_i \in \mathcal{G}_{\max}$, a quorum $Q_i \subseteq \mathcal{G}_{\max}$ exists such that $Q_i \cap F = \emptyset$. It follows that for every $p_i \in \mathcal{G}_{\max}$, there is a set $F_i \in \mathcal{F}_i$ such that $F \subseteq F_i$. Recall that since $\mathbb{Q}$ is a quorum system, $B^3(\mathcal{F})$ holds. From Definition 4.2, we have that for all $i, j \in [1, n]$, $\forall F_i \in \mathcal{F}_i, \forall F_j \in \mathcal{G}_j, \forall F_{ij} \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^* : \mathcal{P} \not\subseteq F_i \cup F_j \cup F_{ij}$.

Towards a contradiction, assume that there is a process p_j such that there exists a quorum $Q_j \in \mathcal{Q}_j$ for p_j with $Q_j = F$. This implies that there exists $F_j \in \mathcal{F}_j$ such that $F_j = \mathcal{P} \setminus F$.

Let F_i be the fail-prone system of $p_i \in \mathcal{G}_{\max}$ such that $F \subseteq F_i$ and let $F_j = \mathcal{P} \setminus F$ as just defined. Then, $F_i \cup F_j \cup F_{ij} = \mathcal{P}$. This follows from the fact that F_i contains F and $F_j = \mathcal{P} \setminus F$. This contradicts the B^3-condition for $\mathbb{F}$. □

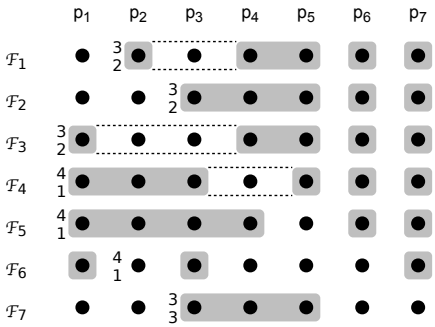
Lemma 4.7. *Let $\mathcal{G}_{max}$ be the maximal guild for a given execution and let p_i be any correct process. Then, every quorum for p_i contains at least one process in $\mathcal{G}_{max}$.*

Proof. The claim naturally derives from the consistency property of an asymmetric Byzantine quorum system. Consider any correct process p_i and one of its quorums, $Q_i \in \mathcal{Q}_i$. For any process $p_j \in \mathcal{G}_{\max}$, let Q_j be a quorum of p_j such that $Q_j \subseteq \mathcal{G}_{\max}$, which exists because $\mathcal{G}_{\max}$ is a guild. Then, the quorum consistency property implies that $Q_i \cap Q_j \neq \emptyset$. Thus, Q_i contains a process in the maximal guild. □

Finally, we show with an example that it is possible for a wise process to be outside the maximal guild.

Example 4.8. *Let us consider a seven-process asymmetric Byzantine quorum system $\mathbb{Q}_A$, defined through its fail-prone system $\mathbb{F}_A$. The notation $\Theta_k^n(\mathcal{C})$ for a set $\mathcal{C}$ with n elements denotes the* threshold *combination operator and enumerates all subsets of $\mathcal{C}$ of cardinality k. The diagram below shows fail-prone sets as shaded areas and the notation $\binom{n}{k}$ in front of a fail-prone set stands for k out of the n processes in the set.*

$\mathbb{F}_A$:



One can verify that $B^3(\mathbb{F}_A)$ holds; hence, let $\mathbb{Q}_A$ be the canonical asymmetric Byzantine quorum system of $\mathbb{F}_A$.

$\mathbb{Q}_A$:

$$
\begin{aligned}
\mathcal{Q}_1 &= \{\{p_1, p_3, p_5\}, \{p_1, p_3, p_4\}, \{p_1, p_2, p_3\}\} \\
\mathcal{Q}_2 &= \{\{p_1, p_2, p_5\}, \{p_1, p_2, p_4\}, \{p_1, p_2, p_3\}\} \\
\mathcal{Q}_3 &= \{\{p_2, p_3, p_5\}, \{p_2, p_3, p_4\}, \{p_1, p_2, p_3\}\} \\
\mathcal{Q}_4 &= \{\{p_1, p_2, p_3, p_4\}, \{p_1, p_2, p_4, p_5\}, \{p_1, p_3, p_4, p_5\}, \\
&\quad \{p_2, p_3, p_4, p_5\}\} \\
\mathcal{Q}_5 &= \{\{p_1, p_2, p_3, p_5\}, \{p_1, p_2, p_4, p_5\}, \{p_1, p_3, p_4, p_5\}, \\
&\quad \{p_2, p_3, p_4, p_5\}\} \\
\mathcal{Q}_6 &= \{\{p_2, p_4, p_5, p_6\}\} \\
\mathcal{Q}_7 &= \{\{p_1, p_2, p_6, p_7\}\}
\end{aligned}
$$

With $F = \{p_4, p_5\}$, for instance, processes p_1, p_2, p_3 and p_7 are wise, p_6 is naïve, and the maximal guild is $\mathcal{G}_{\max} = \{p_1, p_2, p_3\}$. It follows that process p_7 is wise but outside the guild $\mathcal{G}_{\max}$, because quorum $Q_7 \in \mathcal{Q}_7$ contains the naïve process p_6.

Lemma 4.7 reveals the interesting result that for an execution with a guild, each quorum of every correct process p_i contains at least one process that is also in the maximal guild $\mathcal{G}_{\max}$. Since a kernel for p_i is a process set that has some member in common with every quorum of p_i, this implies that $\mathcal{G}_{\max}$ is a kernel for p_i.

Corollary 4.9. *In every execution with a guild, the maximal guild $\mathcal{G}_{max}$ is a kernel for every correct process.*

It follows that whenever all processes in the maximal guild send some particular message, then every correct process will eventually receive this message from all processes in one of its kernels. This is exploited by protocols that use kernels.

A guild can also be seen as a set of sufficiently many wise processes that allow a protocol to make progress, in the following sense.

Lemma 4.10. *Consider an execution, in which the processes in F are faulty and let $\mathcal{G}_{max}$ be the maximal guild for F. Let A be a superset of F that is disjoint from $\mathcal{G}_{max}$, i.e., $F \subseteq A \subseteq \mathcal{P} \setminus \mathcal{G}_{max}$.*

Then, in any execution where the processes in A fail, $\mathcal{G}_{max}$ is also the maximal guild for A.

Proof. Let $\mathcal{G}_{\max}$ be the maximal guild in an execution with set of faulty processes $F \subseteq \mathcal{P} \setminus \mathcal{G}_{\max}$. By definition of a guild, $\mathcal{G}_{\max}$ contains a quorum for each of its members. This means that there exists a quorum Q_i for every $p_i \in \mathcal{G}_{\max}$ such that $Q_i \cap F = \emptyset$. This also implies that for every set $A \supseteq F$, with $A \subseteq \mathcal{P} \setminus \mathcal{G}_{\max}$, we have that $Q_i \cap A = \emptyset$, and the lemma follows. □

4.2 Composition of asymmetric Byzantine quorum systems

We now explore the problem of composing trust assumptions, as expressed by Byzantine quorum systems. Starting from two or more running distributed systems, each one with its own assumption, how can they be combined, so that their participant groups are joined and operate together? A simple, but not so intriguing solution could be to stop all running protocols and to redefine the trust structure from scratch, with full knowledge of all assumptions across the participants. With symmetric trust, a new global assumption that includes all participants would be defined. In the asymmetric-trust model, every process would specify new personal assumptions on all other participants. Subsequently, the composite system would have to be restarted. Although this solution can be effective, it requires that all members of each initial group express assumptions about the trustworthiness of the processes in the other groups. In realistic scenarios, this might not be possible, since the participants of one system lack knowledge about the members of other systems, and can therefore not express their trust about them. Moreover, one needs to ensure that the combined system satisfies the liveness

and safety conditions, as expressed by the B^3-condition for quorum intersection. Since the assumptions are personal, it is not guaranteed, and in practice quite challenging, that the composite system will indeed satisfy the B^3-condition.

Given two ABQS, $\mathbb{Q}_1$ defined on processes $\mathcal{P}_1$ with fail-prone system $\mathbb{F}_1$, and $\mathbb{Q}_2$ defined on processes $\mathcal{P}_2$ with fail-prone system $\mathbb{F}_2$, we want to provide a *composition* rule that allows the processes $\mathcal{P}_3 = \mathcal{P}_1 \cup \mathcal{P}_2$ to form an ABQS $\mathbb{Q}_3$ with fail-prone system $\mathbb{F}_3$.

4.2.1 The tolerated system of an ABQS

For defining composition with ABQS, we first introduce the central notion of the *tolerated system* of an ABQS. Recall that in a symmetric-trust setting, the processes agree on the possible failures, that is, on which processes might crash or collaborate to break security. In an asymmetric-trust setting, no such common understanding exists, either because there is not enough knowledge to make such an assumption on the system, or because the participants simply do not agree with each other. In this model, every process expresses its own beliefs and expectations, and no global notion of "correct" belief exists. In every execution, however, there will be a ground truth, manifested by a set of actually faulty processes, and not all members of the system will have correctly anticipated this ground truth. Again, since there is no global understanding of the world, this is expected to happen. However, the participants might still be able to make progress (where progress is defined by the protocol they are running), exactly in those executions when a guild exists.

An external process examining an asymmetric-trust system without any prior knowledge or beliefs about the processes cannot assess the trust assumptions of any individual process. However, the third process can evaluate the system based on its ability to make progress through a guild.

The central concept for composing two ABQS is the *tolerated system* of an ABQS. Recall that in an execution where all processes in $F \subset \mathcal{P}$ actually fail, there may also be naïve processes, wise processes that form a guild $\mathcal{G}$, and wise processes outside the guild (Example 4.8). For a specific guild $\mathcal{G} \neq \emptyset$, the union of all those processes outside $\mathcal{G}$ is called a *tolerated set* because the guild is autonomous without any of them. Hence, the tolerated set consists of the faulty, the naïve, and the wise processes outside the guild. The tolerated system contains all the tolerated sets. Formally, we have the following definition.

Definition 4.11 (Tolerated system)**.** *Given an asymmetric Byzantine quorum system $\mathbb{Q}$, a set of processes T is called* tolerated (by $\mathbb{Q}$) *if in any execution with faulty processes F, a guild $\mathcal{G}$ for F and $\mathbb{Q}$ exists such that $T = \mathcal{P} \setminus \mathcal{G}$.*

The tolerated system $\mathcal{T}$ *of an asymmetric Byzantine quorum system $\mathbb{Q}$ is the collection of all maximal tolerated sets.*

Intuitively, the tolerated system of an ABQS reflects the resilience of the ABQS: even without the processes in a tolerated set, there still exists a guild. Therefore, the tolerated system characterizes the executions in which some of the processes in the asymmetric system will be able to operate correctly and make progress. In that sense, the tolerated system of an ABQS is the counterpart of the fail-prone system for a symmetric Byzantine quorum system.

Notice that the tolerated system is a global notion emerging from the subjective trust choices of the processes; any process that knows the fail-prone and quorum systems of all processes can calculate it. We show later that the tolerated systems of two ABQS play a crucial role for composing them; the processes in the first system will use the tolerated sets of the second system as their trust assumptions, and vice versa. Consequently, the processes in the first system only need to know the tolerated system of the second system.

The following lemma shows that the maximal tolerated system of a canonical ABQS naturally corresponds to a (global) fail-prone system among all the processes.

Lemma 4.12. *Let $\mathbb{Q}$ be an ABQS on processes $\mathcal{P}$ with asymmetric fail-prone system $\mathbb{F} = \overline{\mathbb{Q}}$, i.e., such that $\mathbb{Q}$ is a canonical ABQS. Then the tolerated system $\mathcal{T}$ of $\mathbb{Q}$ is a (global) fail-prone system among all the processes in $\mathcal{P}$. In particular, if $B^3(\mathbb{F})$, then $Q^3(\mathcal{T})$.*

Proof. Towards a contradiction, let us assume that $\mathcal{T}$ does not satisfy the Q^3-condition. This means that there exist $T_1, T_2, T_3 \in \mathcal{T}$ such that $T_1 \cup T_2 \cup T_3 = \mathcal{P}$. Also, let $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$ be the corresponding guilds, i.e., $\mathcal{G}_1 = \mathcal{P} \setminus \mathcal{T}_1, \mathcal{G}_2 = \mathcal{P} \setminus \mathcal{T}_2$ and $\mathcal{G}_3 = \mathcal{P} \setminus \mathcal{T}_3$.

Without loss of generality every guild contains at least a process, and at least a quorum for this process is fully contained in the guild. By the consistency property of an ABQS, these quorums must intersect pairwise, hence the guilds also intersect pairwise. This means that there exist processes $p_i \in \mathcal{G}_1 \cap \mathcal{G}_2$ and $p_j \in \mathcal{G}_2 \cap \mathcal{G}_3$. Now, because p_i is a member of $\mathcal{G}_1$, we can make the following reasoning: p_i has a quorum

$Q_i \in \mathcal{Q}_i$ such that $Q_i \subseteq \mathcal{G}_1$, the Byzantine quorum system is canonical, so p_i has a fail-prone set $F_i = \mathcal{P} \setminus Q_i \in \mathcal{F}_i$, thus we get $T_1 \subseteq F_i$, i.e., $T_1 \in \mathcal{F}_i$. With similar reasoning, we get $T_2 \in \mathcal{F}_i$ (because $p_i \in \mathcal{G}_2$), $T_2 \in \mathcal{F}_j$ (because $p_j \in \mathcal{G}_2$), and $T_3 \in \mathcal{F}_j$ (because $p_j \in \mathcal{G}_3$). But this is a contradiction, because p_i and p_j with fail-prone sets T_1, T_2, and T_3 violate the B^3-condition in $\mathbb{Q}$. □

As has been known before, by Lemma 3.3, if $\mathcal{T}$ satisfies the Q^3-condition, then there exists also a symmetric Byzantine quorum system for the fail-prone system $\mathcal{T}$; for instance, this may be the canonical Byzantine quorum system $\overline{\mathcal{T}}$.

Lemma 4.12 confirms the intuition that the tolerated set of an ABQS is the counterpart of a fail-prone set in a symmetric-trust system.

4.2.2 How clients interact with an ABQS

Many practical replication protocols separate clients from replicas; in state-machine replication, clients submit commands, replicas totally order and execute them, and then send back responses to the clients. When the expected failures among replicas are modeled as a Byzantine quorum system, that is, with a symmetric trust assumption, the clients wait for responses from a quorum of replicas. However, if the trust assumption among the replicas is asymmetric, it is unclear which sets of participants are capable to convince a client to accept a response. The subjective quorums of the replicas only express their personal beliefs, which the clients may not share.

One way to resolve this could be to let each client express trust in the replicas through its own quorum system. But if clients do not have sufficient knowledge to make such assumptions, they need a global property of the quorum system to decide on its responses, and this can be the tolerated system. Note that every guild formed by replicas corresponds to the complement of a tolerated set. This indicates that (at least some) replicas did agree on their trustworthiness, and this may convince the client. Indeed, we will use this idea in the composition procedure for ABQS. Specifically, the participants of each system may operate as clients of the other and could send a composition-request message, waiting for responses from a guild of participants.

4.2.3 Composition of ABQS

Based on the remarks above, we claim that any form of composition between two ABQS must satisfy the following conditions. Regarding notation, we want to compose $\mathbb{Q}_1$ with $\mathbb{Q}_2$, resulting in $\mathbb{Q}_3$, with respective asymmetric fail-prone systems $\mathbb{F}_1$, $\mathbb{F}_2$, and $\mathbb{F}_3$. For $k = 1, 2$ and for any $p_i \in \mathcal{P}_k$, let $\mathcal{F}_i^{(k)}$ be the fail-prone system of p_i in $\mathbb{F}_k$, and $\mathcal{F}_i^{(3)}$ the fail-prone system of p_i in the resulting $\mathbb{F}_3$. Moreover, let $\mathcal{T}_k$ be the tolerated system of $\mathbb{Q}_k$.

In the text below, the notation $\mathcal{X}|_{\mathcal{P}}$ denotes the restriction of a set $\mathcal{X}$ to $\mathcal{P}$.

1. If $p_i \in \mathcal{P}_1$ and $p_i \in \mathcal{P}_2$, then any $F_i \in \mathcal{F}_i^{(3)}$ must respect the trust assumptions of p_i in $\mathcal{P}_1$ and in $\mathcal{P}_2$, i.e., it must satisfy $F_i|_{\mathcal{P}_1} \in {\mathcal{F}_i^{(1)}}^*$ and $F_i|_{\mathcal{P}_2} \in {\mathcal{F}_i^{(2)}}^*$. If p_i is only in $\mathcal{P}_1$ (and the same holds for $\mathcal{P}_2$), then any $F_i \in \mathcal{F}_i^{(3)}$ must respect the assumptions of p_i in $\mathcal{P}_1$, i.e., $F_i|_{\mathcal{P}_1} \in {\mathcal{F}_i^{(1)}}^*$, and $F_i|_{\mathcal{P}_2}$ can only be one of the tolerated sets in $\mathcal{P}_2$, i.e., $F_i|_{\mathcal{P}_2} \in {\mathcal{T}_2}^*$, since p_i has no assumptions for $\mathcal{P}_2$.

2. If the B^3-condition holds for $\mathbb{F}_1$ and for $\mathbb{F}_2$, then it also holds for composite system, for $\mathbb{F}_3$.

3. For any $p_i \in \mathcal{P}_3$ and any $F_i \in \mathcal{F}_i^{(3)}$, there exists a quorum $Q_i \in \mathcal{Q}_i^{(3)}$, such that $F_i \cap Q_i = \emptyset$.

4. **Preserving wisdom.** In all executions, where there exists a guild $\mathcal{G}_1$ in $\mathbb{Q}_1$ and a guild $\mathcal{G}_2$ in $\mathbb{Q}_2$, the processes in $\mathcal{G}_1 \cup \mathcal{G}_2$ will form a guild in $\mathbb{Q}_3$. The intuition is that, given an execution with F as faulty set, if a process correctly foresees F (and thus enjoys the properties of a guild) in its own system, and if there is a guild in the other system, then this process should also enjoy the properties of a guild in the composite system.

5. **Reducibility to symmetric.** If all processes have the same trust assumptions (in which case $\mathbb{Q}_1$ and $\mathbb{Q}_2$ reduce to symmetric Byzantine quorum systems), then the composite system $\mathbb{Q}_3$ is a symmetric Byzantine quorum system.

Alpos, Cachin, and Zanolini [3] study the composition of (symmetric) Byzantine quorum systems. Their findings result in a deterministic

composition rule among Byzantine quorum systems (and symmetric fail-prone systems) that guarantees that consistency and availability properties are satisfied in the composite system.

Lemma 4.13. *Property 1 implies Property 5.*

Proof. This follows immediately by observing that when all processes in $\mathcal{P}_k$ have the same fail-prone system $\mathcal{F}_k$, for $k = 1, 2$, then the tolerated system $\mathcal{T}_k$ is $\mathcal{F}_k$ itself. Then, Property 1 implies that $\mathcal{F}_i^{(3)}$ is the same for every $p_i \in \mathcal{P}_3$, and that every $F \in \mathcal{F}_i^{(3)}$ satisfies $F|_{\mathcal{P}_1} \in {\mathcal{F}_1}^*$ and $F|_{\mathcal{P}_2} \in {\mathcal{F}_2}^*$. □

Now let us consider two ABQS $\mathbb{Q}_1$ and $\mathbb{Q}_2$ on processes $\mathcal{P}_1$ and $\mathcal{P}_2$ with asymmetric fail-prone systems $\mathbb{F}_1$ and $\mathbb{F}_2$, respectively. All processes in $\mathcal{P}_1$ and $\mathcal{P}_2$ wish to jointly run a protocol, without making any extra assumption about the participants of the other set. Intuitively speaking, each set of processes might have their own issues, their own agreements and disagreements, their own good and bad executions, but they still want to work together. As reasoned earlier, each process in $\mathcal{P}_1$ is an external observer for $\mathcal{P}_2$. Hence, the best a participant in $\mathcal{P}_1$ can do, assuming they have no knowledge, beliefs, or assumptions for the processes of the second set, is to use the tolerated system of $\mathbb{Q}_2$. The same applies, of course, for processes in $\mathcal{P}_2$ with respect to $\mathcal{P}_1$. This leads to the composition procedure we describe next.

Construction 4.14 (Purification)**.** *Let $\mathbb{Q}$ an ABQS on processes $\mathcal{P} = \{p_1, \ldots, p_n\}$, with asymmetric fail-prone system $\mathbb{F} = \{\mathcal{F}_1, \ldots, \mathcal{F}_n\}$, such that $B^3(\mathbb{F})$, and let $\mathcal{T}$ its tolerated system. Assume $Q^3(\mathcal{T})$. As we have seen, this is always the case for canonical ABQS. We want to* purify $\mathbb{F}$ *so that $B^3([\mathcal{F}_1, \ldots, \mathcal{F}_n, \mathcal{T}])$, i.e., $\forall F_i \in \mathcal{F}_i, \forall F_j \in \mathcal{T}, \forall F_{ij} \in \mathcal{F}_i^* \cap \mathcal{T}^*$ it holds that $\mathcal{P} \not\subseteq F_i \cup F_j \cup F_{ij}$. To do so, every process p_i evaluates the B^3-condition including $\mathcal{T}$ in the asymmetric fail-prone system $\mathbb{F}$. If it does not hold, then for any $F_i \in \mathcal{F}_i$ that violates the B^3-condition, p_i removes F_i from $\mathcal{F}_i$, and adds to $\mathcal{F}_i$ all those subsets of F_i that do not violate the B^3-condition. This results in a* purified *fail-prone system, which, by construction, satisfies the B^3-condition.*

Intuitively, the purification procedure removes fail-prone systems that are "useless," in the sense that they do not influence the existence of a guild, as shown by the next lemma. Seen from a higher level, it is an expression of the fact that processes have their own beliefs, but also need to adapt to those of the others; a process p_i might expect a set F to fail

during an execution and construct its fail-prone system $\mathcal{F}_i$ so as to be protected against F. However, if the beliefs of other processes are such that the failure of F does not lead to a guild, i.e., F is not tolerated, then p_i can not benefit from including F in $\mathcal{F}_i$.

Lemma 4.15. *For every possible execution with a guild $\mathcal{G}$, a process in $\mathcal{G}$ of the non-purified system is also contained in some guild of the purified system.*

Proof. Observe that the $F_i \in \mathcal{F}_i$ which p_i removes cannot be in $\mathcal{T}$, because otherwise it would be possible to cover all $\mathcal{P}$ with sets in $\mathcal{T}$; but this is not possible by the assumption $Q^3(\mathcal{T})$. This implies that the failure of F_i cannot lead to the existence of a guild, and can be removed from $\mathcal{F}_i$. On the other hand, subsets of F_i can possibly be in $\mathcal{T}$, and p_i keeps those subsets in $\mathcal{F}_i$. □

Observe that the purification procedure is deterministic and uses information that is available to every process in the system: evaluating the B^3-condition, for example, already assumes that every process in the system knows the asymmetric fail-prone systems of the others and that Byzantine processes do not lie about their assumptions.

We define the *cartesian operator*, which will be used in our composition rule below.

Definition 4.16. *Let $\mathcal{A} = \{A_1, \ldots, A_m\}$ and $\mathcal{B} = \{B_1, \ldots, B_n\}$ be two sets of subsets of $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively. We define $\mathcal{A} \otimes \mathcal{B}$ as the set that contains the union of all sets $A_i \in \mathcal{A}^*$ and $B_j \in \mathcal{B}^*$, under the restriction that either both A_i and B_j contain exactly the same subset of the processes common to $\mathcal{P}_1$ and $\mathcal{P}_2$ or they do not have anything in common. Formally,*

$$\mathcal{A} \otimes \mathcal{B} = \big\{ A_i \cup B_j \mid A_i \in \mathcal{A}^* \wedge B_j \in \mathcal{B}^* \wedge (\forall C \subseteq \mathcal{P}_1 \cap \mathcal{P}_2 : C \subseteq A_i \Leftrightarrow C \subseteq B_j) \big\}.$$

We have the following intermediate result.

Lemma 4.17. *Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be two fail-prone systems of the sets of processes $\mathcal{P}_1$ and $\mathcal{P}_2$, respectively, where $\mathcal{P}_1$ and $\mathcal{P}_2$ might contain common processes. If $Q^3(\mathcal{F}_1)$ and $Q^3(\mathcal{F}_2)$, then for $\mathcal{F}_3 = \mathcal{F}_1 \otimes \mathcal{F}_2$ as a fail-prone system of $\mathcal{P}_3 = \mathcal{P}_1 \cup \mathcal{P}_2$ it holds $Q^3(\mathcal{F}_3)$.*

Proof. Any $F \in \mathcal{F}_3$ either does not contain a set of common processes C among $\mathcal{P}_1$ and $\mathcal{P}_2$ or it does. In the former case, it is immediate to see that $F|_{\mathcal{P}_1} \in \mathcal{F}_1^*$ and $F|_{\mathcal{P}_2} \in \mathcal{F}_2^*$. In the latter case, F has been created as the union between $F_i \in \mathcal{F}_1^*$ and $F_j \in \mathcal{F}_2^*$, both containing the same subset of $P_1 \cap P_2$. It is thus not possible that a new element of $\mathcal{P}_1$ appears in $F|_{\mathcal{P}_1}$ that was not already in F_i, and similarly that a new element of $\mathcal{P}_2$ appears in $F|_{\mathcal{P}_2}$ that was not already in F_j. This implies that $F|_{\mathcal{P}_1} \in \mathcal{F}_1^*$ and $F|_{\mathcal{P}_2} \in \mathcal{F}_2^*$. Towards a contradiction, let $F_A, F_B, F_C \in \mathcal{F}_3$ such that $F_A \cup F_B \cup F_C = \mathcal{P}_3$. Now consider the restriction of F_A, F_B and F_C to $\mathcal{P}_1$ (and similarly to $\mathcal{P}_2$). We have that $F_A|_{\mathcal{P}_1} \cup F_B|_{\mathcal{P}_1} \cup F_C|_{\mathcal{P}_1} = \mathcal{P}_1$. However, the sets $F_A|_{\mathcal{P}_1}$, $F_B|_{\mathcal{P}_1}$, and $F_C|_{\mathcal{P}_1}$ are each (subsets of) fail-prone sets in $\mathcal{F}_1$. We thus have found three fail-prone sets that cover $\mathcal{P}_1$, a contradiction to $\mathcal{F}_1$ satisfying the Q^3-condition. □

Construction 4.18 (Composition of ABQS)**.** *Let $\mathcal{P}_1 = \{p_1, \ldots, p_{m+k}\}$ and $\mathcal{P}_2 = \{p_{m+1}, \ldots, p_n\}$ be two sets of processes, with processes $p_{m+1}, \ldots, p_{m+k}$ in common. Let $\mathbb{Q}_1$ be an ABQS on processes $\mathcal{P}_1$ with asymmetric fail-prone system $\mathbb{F}_1 = \{\mathcal{F}_1^{(1)}, \ldots, \mathcal{F}_{m+k}^{(1)}\}$, and $\mathbb{Q}_2$ an ABQS on processes $\mathcal{P}_2$ with asymmetric fail-prone system $\mathbb{F}_2 = \{\mathcal{F}_{m+1}^{(2)}, \ldots, \mathcal{F}_n^{(2)}\}$, where $\mathbb{F}_1$ and $\mathbb{F}_2$ are purified. Moreover, let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the tolerated systems of the two ABQS, respectively. The* composite fail-prone system $\mathbb{F}_3$ *on processes $\mathcal{P}_3 = \mathcal{P}_1 \cup \mathcal{P}_2$ is*

$$\mathbb{F}_3 = [\mathcal{F}_1^{(1)} \otimes \mathcal{T}_2, \ldots, \mathcal{F}_m^{(1)} \otimes \mathcal{T}_2, \mathcal{F}_{m+1}^{(1)} \otimes \mathcal{F}_{m+1}^{(2)}, \ldots, \\ , \ldots, \mathcal{F}_{m+k}^{(1)} \otimes \mathcal{F}_{m+k}^{(2)}, \mathcal{F}_{m+k+1}^{(2)} \otimes \mathcal{T}_1, \ldots, \mathcal{F}_n^{(2)} \otimes \mathcal{T}_1].$$

and the composite ABQS $\mathbb{Q}_3$ *is any asymmetric Byzantine quorum system for $\mathbb{F}_3$.*

Lemma 4.19. *The composed fail-prone system $\mathbb{F}_3$ resulting from Construction 4.18 satisfies the B^3-condition.*

Proof. Towards a contradiction, let us assume that the B^3-condition does not hold on $\mathbb{F}_3$. This means there exist processes p_i and p_j and fail-prone sets $F_i \in \mathcal{F}_i^{(3)}$, $F_j \in \mathcal{F}_j^{(3)}$, and $F_{ij} \in \mathcal{F}_i^{(3)^*} \cap \mathcal{F}_j^{(3)^*}$ such that $\mathcal{P}_3 = F_i \cup F_j \cup F_{ij}$. In the following we consider the restriction of F_i, F_j, and F_{ij} to $\mathcal{P}_1$, i.e., $F_i|_{\mathcal{P}_1}, F_j|_{\mathcal{P}_1}$, and $F_{ij}|_{\mathcal{P}_1}$, respectively. We distinguish two cases for p_i and p_j. First, consider the case where p_i and p_j belong to different sets of processes and let, w.l.o.g., $p_i \in \mathcal{P}_1 \setminus \mathcal{P}_2$ and $p_j \in \mathcal{P}_2 \setminus \mathcal{P}_1$.

By the definition of the $\otimes$ operator, and with an argument similar to what we used in the proof of Lemma 4.17, we get that $F_i|_{\mathcal{P}_1} \in {\mathcal{F}_i^{(1)}}^*$, that $F_j|_{\mathcal{P}_1} \in {\mathcal{T}_1}^*$, that $F_{ij}|_{\mathcal{P}_1}$ is a common subset of ${\mathcal{F}_i^{(1)}}^*$ and ${\mathcal{T}_1}^*$, and that their union covers $\mathcal{P}_1$. This is a contradiction because $\mathbb{F}_1$ is purified. Second, consider the case where at least one of p_i and p_j belongs to both $\mathcal{P}_1$ and $\mathcal{P}_2$, and let, w.l.o.g., $p_i \in \mathcal{P}_1, p_j \in \mathcal{P}_1 \cap \mathcal{P}_2$. (If $p_i \in \mathcal{P}_1 \cap \mathcal{P}_2$ the same reasoning can be applied by projecting in $\mathcal{P}_2$.) For this case, we observe that $F_i|_{\mathcal{P}_1} \in {\mathcal{F}_i^{(1)}}^*$, $F_j|_{\mathcal{P}_1} \in {\mathcal{F}_j^{(1)}}^*$, and that $F_{ij}|_{\mathcal{P}_1}$ is a common subset of a fail-prone set in $\mathcal{F}_i^{(1)*}$ and a fail-prone set in $\mathcal{F}_j^{(1)*}$. This contradicts the assumption that $B^3(\mathbb{F}_1)$. □

Remark 4.20. *Given an ABQS $\mathbb{Q}_1$ for an asymmetric fail-prone system $\mathbb{F}_1$ on processes $\mathcal{P}_1$, and an ABQS $\mathbb{Q}_2$ for $\mathbb{F}_2$ on $\mathcal{P}_2$, and assuming that the processes of each system make no assumptions about processes in the other, a composition of the two systems is only possible if the corresponding tolerated systems $\mathcal{T}_1$ and $\mathcal{T}_2$ both satisfy the Q^3-condition. This is because the processes of the first system (and vice versa) are only external observers for the second system, and therefore only assess it through its tolerated system. Processes in $\mathcal{P}_1$ want to make sure that whenever the second system is able to make progress (that is, for every $T \in \mathcal{T}_2$ that leads to a guild), they will also be able to make progress. To achieve this, they must consider all tolerated sets $T \in \mathcal{T}_2$ as a possible failed set. However, because the processes of the first system do not assume anything about the second system, the only way to achieve this is to include all $T \in \mathcal{T}_2$ in their fail-prone sets. This leads to an ABQS if and only if the Q^3-condition holds in the second system (and vice versa).*

Lemma 4.19 and Theorem 4.3 together imply the existence of an ABQS for $\mathbb{F}_3$ as defined in Construction 4.18. This is the asymmetric canonical quorum system $\mathbb{Q}_3 = \overline{\mathbb{F}}_3$.

For instance, let us consider two ABQS $\mathbb{Q}_1$ and $\mathbb{Q}_2$ on processes $\mathcal{P}_1 = \{p_1, \ldots, p_m\}$ and $\mathcal{P}_2 = \{p_{m+1}, \ldots, p_n\}$ with asymmetric fail-prone systems $\mathbb{F}_1$ and $\mathbb{F}_2$, respectively, such that $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$. Then, the asymmetric canonical quorum system for $\mathbb{F}_3$ is

$$\mathbb{Q}_3 = [\mathcal{Q}_1 \cup \overline{\mathcal{T}}_2, \ldots, \mathcal{Q}_m \cup \overline{\mathcal{T}}_2, \mathcal{Q}_{m+1} \cup \overline{\mathcal{T}}_1, \ldots, \mathcal{Q}_n \cup \overline{\mathcal{T}}_1],$$

where $\mathcal{Q}_i = \overline{\mathcal{F}}_i$, $\mathcal{Q}_i \cup \overline{\mathcal{T}}_j = \{Q_k \cup \mathcal{G}_l \mid Q_k \in \mathcal{Q}_i \wedge \mathcal{G}_l \in \overline{\mathcal{T}}_j\}$ and $\mathcal{G}_i$ is a guild for a tolerated set in $\mathcal{T}_j$. Notice that, by definition, $\overline{\mathcal{T}}$ contains all the guilds that can be obtained within an ABQS.

As a short proof of why $\mathbb{Q}_3$ is the canonical asymmetric Byzantine quorum system of $\mathbb{F}_3$, we observe that, by assuming $\mathcal{P}_1 \cap \mathcal{P}_2 = \emptyset$, the asymmetric fail-prone system $\mathbb{F}_3$ in Construction 4.18 reduces to

$$\mathbb{F}_3 = [\mathcal{F}_1 \cup \mathcal{T}_2, \ldots, \mathcal{F}_m \cup \mathcal{T}_2, \mathcal{F}_{m+1} \cup \mathcal{T}_1, \ldots, \mathcal{F}_n \cup \mathcal{T}_1],$$

where $\mathcal{F}_i \cup \mathcal{T}_j = \{F_k \cup T_l \mid F_k \in \mathcal{F}_i \wedge T_l \in \mathcal{T}_j\}$. If we consider the bijective complement of $\mathcal{F}_i \cup \mathcal{T}_j$ this is made by all the sets of the form $\overline{F_k \cup T_l}$ in $\mathcal{P}_3 = \mathcal{P}_1 \cup \mathcal{P}_2$. Then, $\overline{F_k \cup T_l} = \overline{F}_k \cap \overline{T}_l = (Q_k \cup \mathcal{P}_2) \cap (\mathcal{G}_l \cup \mathcal{P}_1)$ where $Q_k = \overline{F}_k$ in $\mathcal{P}_1$. Finally, $(Q_k \cup \mathcal{P}_2) \cap (\mathcal{G}_l \cup \mathcal{P}_1) = (Q_k \cap \mathcal{G}_l) \cup (Q_k \cap \mathcal{P}_1) \cup (\mathcal{P}_2 \cap \mathcal{G}_l) \cup (\mathcal{P}_2 \cap \mathcal{P}_1)$. Observe that, by assumption on the sets of processes, it follows that $(\mathcal{P}_2 \cap \mathcal{P}_1) = \emptyset$ and $(Q_k \cap \mathcal{G}_l) = \emptyset$. So, $\overline{F_k \cup T_l} = (Q_k \cap \mathcal{P}_1) \cup (\mathcal{P}_2 \cap \mathcal{G}_l) = Q_k \cup \mathcal{G}_l$.

4.2.4 Composition in practice

We now sketch a protocol that can be used by two (possibly disjoint) sets of processes $\mathcal{P}_1$ and $\mathcal{P}_2$ that form two asymmetric Byzantine quorum systems $\mathbb{Q}_1$ and $\mathbb{Q}_2$ with asymmetric fail-prone systems $\mathbb{F}_1$ and $\mathbb{F}_2$, respectively. We assume that processes in $\mathcal{P}_1$ and $\mathcal{P}_2$ are running two different instances of the same Byzantine consensus protocol (i.e., providing total-order broadcast) and that $\mathbb{F}_1$ and $\mathbb{F}_2$ are publicly known.

The composition can be initiated by any process p_i in $\mathcal{P}_1$. To that end, process p_i, acting as a client for $\mathbb{Q}_1$, sends a *composition-request* message to every process in $\mathcal{P}_2$. Upon receiving this request, processes in $\mathcal{P}_2$ start a round of Byzantine consensus: if a sufficient number of processes vote for the composition, it will be agreed. Assume the protocol instance run by $\mathbb{Q}_2$ has a history of delivered messages $\mathcal{H}_2$ at this point. Then, upon deciding, processes in $\mathcal{P}_2$ send a *composition-response* message, which includes $\mathcal{H}_2$, back to $\mathcal{P}_1$.

The rest of the protocol is symmetric to the first part; any process in $\mathcal{P}_1$ that receives the same composition response from a guild of $\mathcal{P}_2$ participates in a round of Byzantine consensus, this time within $\mathcal{P}_1$. This results in $\mathcal{P}_1$ sending a *composition-acknowledgment* message to $\mathcal{P}_2$, which now includes $\mathcal{H}_1$, the history of delivered messages in the instance run by $\mathcal{P}_1$. The histories $\mathcal{H}_1$ and $\mathcal{H}_2$ can be used by the composed system to calculate the initial state of the new protocol instance, presumably using a generic *merge function*.

The composition-acknowledgment message signals the start of a new protocol instance run by $\mathcal{P}_1 \cup \mathcal{P}_2$. From this point on, processes use the

composed fail-prone and quorum systems. Since $\mathbb{F}_2$ is known, processes in $\mathcal{P}_1$ can calculate both the tolerated system $\mathcal{T}_2$ of $\mathbb{Q}_2$ (in the simplest case by trying all possible failures of $\mathcal{P}_2$) and the purified version of $\mathbb{F}_2$, and vice versa for processes in $\mathcal{P}_2$. Should the fail-prone systems not be public, the processes could send them in the composition messages; however, privacy aspects are beyond the focus of this work.

Chapter 5

Asymmetric Byzantine Consensus

Consensus represents a fundamental abstraction in distributed systems. It captures the problem of reaching agreement among multiple processes on a common value, despite unreliable communication and the presence of faulty processes. Most protocols for consensus operate under the assumption that the *number* of faulty processes is limited. Moreover, all processes in the system share this common trust assumption. Since the advent of blockchains systems, as we saw in Chapter 4, more flexible trust models have been introduced, opening up the possibility to implement consensus with subjective trust.

In this chapter we define asymmetric Byzantine consensus. Then we implement it by a randomized algorithm, which is based on the protocol of Mostéfaoui, Moumen, and Raynal [30] as presented in Section 5.2. Our implementation also fixes the problem described there. Our notion of Byzantine consensus, which is formally presented in Section 5.3, uses strong validity in the asymmetric model. Furthermore, it restricts the safety properties of consensus from all correct ones to *wise* processes. For implementing asynchronous consensus, we use a system enriched with randomization. In round-based consensus algorithms, the termination property is formulated with respect to the round number r that a process executes. The corresponding probabilistic termination property can be guaranteed only for wise processes.

Finally, we conclude this chapter by precisely presenting a problem

that may arise in the asymmetric-trust model with leader-based consensus protocols such as PBFT [13] or HotStuff [38]. We analyze such problem in Section 5.4 by means of PBFT and we propose a first solution to it. This chapter incorporates content from the papers "Asymmetric asynchronous byzantine consensus" [10] and "Brief announcement: Revisiting signature-free asynchronous byzantine consensus" [11].

5.1 System model

Protocol. Let $\mathcal{P}$ be a set of processes as introduced in Chapter 5. A protocol for $\mathcal{P}$ consists of a collection of programs with instructions for all processes. Protocols are presented in a modular way using the event-based notation of Cachin, Guerraoui, and Rodrigues [6].

Functionalities and modularity. A *functionality* is an abstraction of a distributed computation, either used as a primitive available to the processes or defining a service that a protocol run by the processes will provide. Functionalities may be composed in a modular way. Every functionality in the system is specified through its *interface*, containing the *events* that it exposes to applications that may call it, and through a number of *properties* that define its behavior. There are two kinds of events in an interface: *input events* that the functionality receives from other abstractions, typically from an application that invokes its services, and *output events*, through which the functionality delivers information or signals a condition.

Multiple functionalities may be composed together modularly. In a modular protocol implementation, in particular, every process executes the program instructions of the protocol implementations for all functionalities in which it participates.

Links. We assume there is a low-level functionality for sending messages over point-to-point links between each pair of processes. In a protocol, this functionality is accessed through the events of "sending a message" and "receiving a message." Point-to-point messages are authenticated and delivered reliably among correct processes.

Moreover, we assume FIFO ordering on the reliable point-to-point links for every pair of processes (except in Section 5.2). This means that if a correct process has "sent" a message m_1 and subsequently "sent" a message m_2, then every correct process does not "receive" m_2 unless

it has earlier also "received" m_1. FIFO-ordered links are actually a very common assumption. Protocols that guarantee FIFO order on top of (unordered) reliable point-to-point links are well-known and simple to implement [6], [19]. We remark that there is only one FIFO-ordered reliable point-to-point link functionality in the model; hence, FIFO order holds among the messages exchanged by the implementations for all functionalities used by a protocol.

Time and randomization. In this chapter we consider an asynchronous system, where processes have no access to any kind of physical clock, and there is no bound on processing or communication delays. The randomized consensus algorithm delegates probabilistic choices to a *common coin* abstraction [34]; this is a functionality that delivers the same sequence of random binary values to each process, where each binary value has the value 0 or 1 with probability $\frac{1}{2}$.

5.2 Revisiting signature-free asynchronous Byzantine consensus

In 2014, Mostéfaoui, Moumen, and Raynal [30] introduced a round-based asynchronous randomized consensus algorithm for binary values. It had received considerable attention because it was the first protocol with optimal resilience, tolerating up to $f < \frac{n}{3}$ Byzantine processes, that did not use digital signatures. Hence, this protocol needs only authenticated channels and remains secure against a computationally unbounded adversary. Moreover, it takes $O(n^2)$ constant-sized messages in expectation and has a particularly simple structure. Our description here excludes the necessary cost for implementing randomization, for which the protocol relies on an abstract common coin primitive, as defined by Rabin [34].

This protocol, which we call the *PODC-14* version [30] in the following, suffers from a subtle and little-known problem. It may violate liveness, as has been explicitly mentioned by Tholoniat and Gramoli [36]. The corresponding journal publication by Mostéfaoui, Moumen, and Raynal [29], to which we refer as the *JACM-15* version, touches briefly on the issue and goes on to present an extended protocol. This fixes the problem, but requires also many more communication steps and adds considerable complexity.

In this section, we revisit the PODC-14 protocol, point out in detail how it may fail, and introduce a compact way for fixing it. We discovered this issue while extending the algorithm to asymmetric quorums. In Section 5.3, we present the corresponding fixed asymmetric randomized Byzantine consensus protocol and prove it secure. Our protocol changes the PODC-14 version in a crucial way and thereby regains the simplicity of the original approach.

Before addressing randomized consensus, we recall the key abstraction introduced in the PODC-14 paper, a protocol for broadcasting binary values.

5.2.1 Binary validated broadcast

The *binary validated broadcast* primitive has been introduced in the PODC-14 version [30] under the name *binary-value broadcast*.[1] In this primitive, every process may broadcast a bit $b \in \{0,1\}$ by invoking *bv-broadcast*(b). The broadcast primitive outputs at least one value b and possibly also both binary values through a *bv-deliver*(b) event, according to the following notion.

Definition 5.1 (Binary validated broadcast)**.** *A protocol for* binary validated broadcast *satisfies the following properties:*

Validity: *If at least* $(f+1)$ *correct processes* bv-broadcast *the same value* $b \in \{0,1\}$*, then every correct process eventually* bv-delivers b*.*

Integrity: *A correct process* bv-delivers *a particular value* b *at most once and only if* b *has been* bv-broadcast *by some correct process.*

Agreement: *If a correct process* bv-delivers *some value* b*, then every correct process eventually* bv-delivers b*.*

Termination: *Every correct process eventually* bv-delivers *some value* b*.*

The implementation given by Mostéfaoui, Moumen, and Raynal [30] works as follows. When a correct process p_i invokes *bv-broadcast*(b) for $b \in \{0,1\}$, it sends a VALUE message containing b to all processes. Afterwards, whenever a correct process receives VALUE messages containing

[1] Compared to their work, we adjusted some conditions to standard terminology and chose to call the primitive "binary *validated* broadcast" to better emphasize its aspect of validating that a delivered value was broadcast by a correct process.

b from at least $f + 1$ processes and has not itself sent a VALUE message containing b, then it sends such message to every process. Finally, once a correct process receives VALUE messages containing b from at least $2f+1$ processes, it delivers b through *bv-deliver*(b). Note that a process may *bv-deliver* up to two values. A formal description, in the asymmetric model, of this protocol appears in Algorithm 3 in Section 5.3.

5.2.2 Randomized consensus

We recall the notion of *randomized Byzantine consensus* here and its implementation by Mostéfaoui, Moumen, and Raynal [30]. In a consensus primitive, every correct process proposes a value v by invoking *propose*(v), which typically triggers the start of the protocol among processes; it obtains as output a decided value v through a *decide*(v) event. There are no assumptions made about the faulty processes. We use the probabilistic termination property for round-based protocols. It requires that the probability that a correct process decides after executing infinitely many rounds approaches 1.

Definition 5.2 (Strong Byzantine consensus)**.** *A protocol for asynchronous* strong Byzantine consensus *satisfies:*

Probabilistic termination: *Every correct process p_i decides with probability* 1*, in the sense that*

$$\lim_{r \to +\infty} \mathrm{P}[\textit{a correct process } p_i \textit{ decides by round } r] = 1.$$

Strong validity: *A correct process only decides a value that has been proposed by some correct process.*

Integrity: *No correct process decides twice.*

Agreement: *No two correct processes decide differently.*

The probabilistic termination and integrity properties together imply that every correct process decides exactly once, while the agreement property ensures that the decided values are equal. Strong validity asks that if all correct processes propose the same value v, then no correct process decides a value different from v. Otherwise, a correct process may only decide a value that was proposed by some correct process [6]. In a *binary* consensus protocol, as considered here, only 0 and 1 may be proposed.

The implementation of randomized consensus by Mostéfaoui, Moumen, and Raynal [30] delegates its probabilistic choices to a *common coin* abstraction [6], [34], a random source observable by all processes but unpredictable for an adversary. A common coin is invoked at every process by triggering a *release-coin* event. We say that a process *releases* a coin because its value is unpredictable, unless more than f correct processes have invoked the coin. The value $s \in \mathcal{B}$ of the coin with tag r is output through an event *output-coin*.

Definition 5.3 (Common coin)**.** *A protocol for* common coin *satisfies the following properties:*

Termination: *Every correct process eventually outputs a coin value.*

Unpredictability: *Unless more than f correct processes have released the coin, no process has any information about the coin output by a correct process.*

Matching: *With probability 1 every correct process outputs the same coin value.*

No bias: *The distribution of the coin is uniform over $\mathcal{B}$.*

Observe that the unpredictability condition implies that at least $f+1$ *correct* processes are required to release the coin in order for a process to have information about the coin value output by a correct process.

We now recall the implementation of strong Byzantine consensus according to Mostéfaoui, Moumen, and Raynal [30] in the PODC-14 version, shown in Algorithm 1. A correct process *proposes* a binary value b by invoking *rbc-propose*(b); the consensus abstraction *decides* for b through an *rbc-decide*(b) event.

The algorithm proceeds in rounds. In each round, an instance of *bv-broadcast* is invoked. A correct process p_i executes *bv-broadcast* and waits for a value b to be *bv-delivered*, identified by a tag characterizing the current round. When such a bit b is received, p_i adds b to *values* and broadcasts b through an AUX message to all processes. Whenever a process receives an AUX message containing b from p_j, it stores b in a local set $aux[j]$. Once p_i has received a set $B \subseteq values$ of values such that every $b \in B$ has been delivered in AUX messages from at least $n - f$ processes, then p_i releases the coin for the round. Subsequently, the process waits for the coin protocol to output a binary value s through *output-coin*(s), tagged with the current round number.

Algorithm 1 Randomized binary consensus according to Mostéfaoui, Moumen, and Raynal [30] (code for p_i)

```
1: State
2:       round ← 0: current round
3:       values ← {}: set of bv-delivered binary values for the round
4:       aux ← [{}]^n: stores sets of values that have been received in
5:                 AUX messages in the round

6: upon event rbc-propose(b) do
7:       invoke bv-broadcast(b) with tag round

8: upon bv-deliver(b) with tag r such that r = round do
9:       values ← values ∪ {b}
10:      send message [AUX,round,b] to all p_j ∈ P

11: upon receiving a message [AUX,r,b] from p_j such that r = round do
12:      aux[j] ← aux[j] ∪ {b}

13: upon exists B ⊆ values such that B ≠ {} and
14:           |{p_j ∈ P | B = aux[j]}| ≥ n − f do
15:      release-coin with tag round
16:      wait for output-coin(s) with tag round
17:      round ← round + 1
18:      if exists b such that B = {b} then                  // i.e., |B| = 1
19:           if b = s then
20:                output rbc-decide(b)
21:           invoke bv-broadcast(b) with tag round
22:           // propose b for the next round
23:      else
24:           invoke bv-broadcast(s) with tag round
25:           // propose coin value s for the next round
26:      values ← [⊥]^n
27:      aux ← [{}]^n
```

Process p_i then checks if there is a single value b in B. If so, and if $b = s$, then it decides for value b. The process then proceeds to the next round with proposal b. If there is more than one value in B, then p_i changes its proposal to s. In any case, the process starts another round and invokes a new instance of *bv-broadcast* with its proposal. Note that the protocol appears to execute rounds forever.

5.2.3 A liveness problem

Tholoniat and Gramoli [36] mention a liveness issue with the randomized algorithm in the PODC-14 version [30], as presented in the previous section. They sketch a problem that may prevent progress by the correct processes when the messages between them are received in a specific order. In the JACM-15 version, Mostéfaoui, Moumen, and Raynal [29] appear to be aware of the issue and present a different, more complex consensus protocol.

We give a detailed description of the problem in Algorithm 1. Recall the implementation of binary-value broadcast, which disseminates bits in VALUE messages. According to our model, the processes communicate by exchanging messages through an asynchronous reliable point-to-point network. Messages may be reordered, as in the PODC-14 version.

Let us consider a system with $n = 4$ processes and $f = 1$ Byzantine process. Let p_1, p_2, and p_3 be correct processes with input values $0, 1, 1$, respectively, and let p_4 be a Byzantine process with control over the network. Process p_4 aims to cause p_1 and p_3 to release the coin with $B = \{0, 1\}$, so that they subsequently propose the coin value for the next round. If messages are scheduled depending on knowledge of the round's coin value s, it is possible, then, that p_2 releases the coin with $B = \{\bar{s}\}$. Subsequently, p_2 proposes also $\bar{s}$ for the next round, and this may continue forever. We now work out the details, as illustrated in Figures 5.1–5.2.

First, p_4 may cause p_1 to receive $2f + 1$ [VALUE, 1] messages, from p_2, p_3, and p_4, and to *bv-deliver* 1 sent at the start of the round. Then, p_4 sends [VALUE, 0] to p_3, so that p_3 receives value 0 twice (from p_1 and p_4) and also broadcasts a [VALUE, 0] message itself. Process p_4 also sends 0 to p_1, hence, p_1 receives 0 from p_3, p_4, and itself and therefore *bv-delivers* 0. Furthermore, p_4 causes p_3 to *bv-deliver* 0 by making it receive [VALUE, 0] messages from p_1, p_4, and itself. Hence, p_3 *bv-delivers* 0. Finally, process p_3 receives three [VALUE, 1] messages (from itself, p_2, and p_4) and *bv-delivers* also 1.

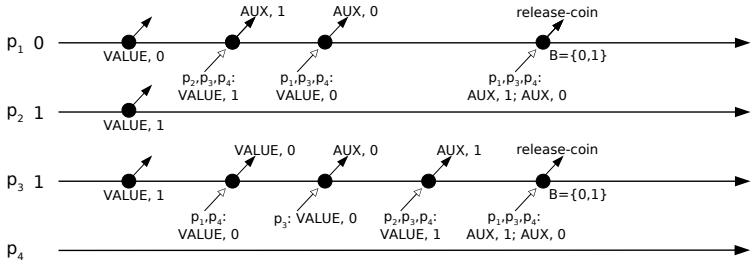


Figure 5.1: The execution of Algorithm 1, where processes p_1 and p_3 execute Line 14 with $B = \{0, 1\}$.

Recall that a process may broadcast more than one AUX message. In particular, it broadcasts an AUX message containing a bit b whenever it has bv-delivered b. Thus, p_1 broadcasts first $[\text{AUX}, 1]$ and subsequently $[\text{AUX}, 0]$, whereas p_3 first broadcasts $[\text{AUX}, 0]$ and then $[\text{AUX}, 1]$. Process p_4 then sends to p_1 and p_3 AUX messages containing 1 and 0. After delivering all six AUX messages, both p_1 and p_3 finally obtain $B = \{0, 1\}$ in Line 14 (Algorithm 1) and see that $|B| \neq 1$ in Line 18 (Algorithm 1). Processes p_1, p_3, and p_4 invoke the common coin.

The Byzantine process p_4 may learn the coin value as soon as p_1 or p_3 have released the common coin, according to unpredictability. Let s be the coin output. We distinguish two cases:

Case $s = 0$: Process p_2 receives now three $[\text{VALUE}, 1]$ messages, from p_3, p_4 and itself, as shown in Figure 5.2. It *bv-delivers* 1 and broadcasts an $[\text{AUX}, 1]$ message. Subsequently, p_2 delivers three AUX messages containing 1, from p_1, p_4 and itself, but no $[\text{AUX}, 0]$ message. It follows that p_2 obtains $B = \{1\}$ and proposes 1 for the next round in Line 21. On the other hand, p_1 and p_3 adopt 0 as their new proposal for the next round, according to Line 24. This means that no progress was made within this round. The three correct processes start the next round again with differing values, again two of them propose one bit and the remaining one proposes the opposite.

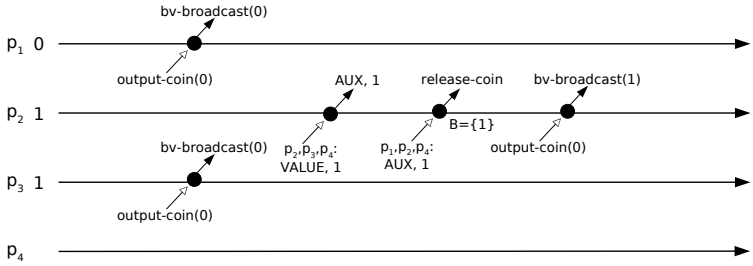


Figure 5.2: Continuing the execution for the case $s = 0$: Process p_2 executes Line 14 with $B = \{1\}$. Processes p_1 and p_3 have already proposed the coin value $s = 0$ for the next round, but p_2 proposes $\overline{s} = 1$.

Case $s = 1$**:** Process p_4 sends $[\text{VALUE}, 0]$ to p_2, so that it delivers two VALUE messages containing 0 (from p_1 and p_4) and thus also broadcast $[\text{VALUE}, 0]$ (this execution is not shown). Recall that p_3 has already sent $[\text{VALUE}, 0]$ before. Thus, p_2 receives $n - f$ $[\text{VALUE}, 0]$ messages, *bv-delivers* 0, and also broadcasts an AUX message containing 0. Subsequently, p_2 may receive $n - f$ messages $[\text{AUX}, 0]$, from p_3, p_4, and itself. It follows that p_2 executes Line 14 with $B = \{0\}$ and chooses 0 as its proposal for the next round (in Line 21). On the other hand, also here, p_1 and p_3 adopt the coin value $s = 1$ and propose 1 for the next round in Line 24. Hence, no progress has been made in this round, as the three correct processes enter the next round with differing values.

The protocol may continue like this forever, producing an infinite execution with no termination.

5.2.4 Fixing the problem

We show how the problem can be prevented with a conceptual insight and two small changes to the original protocol. We do this by recalling the example just presented. The complete protocol and a formal proof are given in Section 5.3, using the more general model of asymmetric quorums.

We start by considering the nature of the common coin abstraction: In any full implementation, the coin is not an abstract oracle, but implemented by a concrete protocol that exchanges messages among the processes.

Observe now that in the problematic execution, the network reorders messages between correct processes. Our first change, therefore, is to assume FIFO ordering on the reliable point-to-point links. This may be implemented over authenticated links, by adding sequence numbers to messages and maintaining a buffer at the receiver [6]. Consider p_2 in the example and the messages it receives from the other correct processes, p_1 and p_3. W.l.o.g. any protocol implementing a common coin requires an additional message exchange, where a correct process sends at least one message to every other process, say, a SHARE message with arbitrary content (to be specific, see Algorithm 2, Section 5.3). Observe that at least $f + 1$ *correct* processes are required to send a SHARE message in order for a process to have information about the coin value.

When p_2 waits for the output of the coin, it needs to receive, again w.l.o.g., a SHARE message from $n - f$ processes. Since the other two correct processes (p_1 and p_3) have sent two VALUE messages and AUX messages each before releasing the coin, then p_2 receives these messages from at least one of them before receiving enough SHARE messages, according to the overlap among Byzantine quorums.

This means that p_2 cannot satisfy the condition in Line 14 with $|B| = 1$. Thus the adversary may no longer exploit its knowledge of the coin value to prevent termination. (Mostéfaoui, Moumen, and Raynal [29] (JACM-15) remark in retrospect about the PODC-14 version that a "fair scheduler" is needed. However, this comes without any proof and thus remains open, especially because the JACM-15 version introduces a much more complex version of the protocol.)

Our second change is to allow the set B to dynamically change while the coin protocol executes. In this way, process p_2 may find a suitable B according to the received AUX messages while concurrently running the coin protocol. Eventually, p_2 will have output the coin *and* its set B will contain the same values as the sets B of p_1 and p_3. Observe that this dynamicity is necessary; process p_2 could start to release the coin after receiving $n - f$ AUX messages containing only the value 1. However, following our example, due to the assumed FIFO order, it will receive from another correct process also an AUX message containing the value 0, before the SHARE message. If we do not ask for the dynamicity of the set B, process p_2, after outputting the coin, will still have

$|B| = 1$. Mostéfaoui, Moumen, and Raynal in the PODC-14 version ([30, Figure 2, Line 5]) seem to rule this out.

Notice that the common-coin primitive here requires *more than f correct* processes to release the coin before it may be predicted by the faulty processes. Within an implementation, this translates into receiving a SHARE message from more than $2f$ processes (or $2f + 1 = n - f$ processes, in case $n = 3f+1$). Abraham, Ben-David, and Yandamuri [1], [2] show that such an assumption (which they call an $2f$-unpredictable coin) is necessary in order to prevent this liveness problem. With an ordinary coin primitive (i.e., one where at least *one correct process* is required to send a SHARE message, before information about the coin value may become available), an adversary would still be able to produce an infinite execution and to violate termination [2, Appendix A].

5.3 Asymmetric randomized Byzantine consensus

In this section we define asymmetric Byzantine consensus. Then we implement it by a randomized algorithm, which is based on the protocol of Mostéfaoui, Moumen, and Raynal [30] as introduced in Section 5.2. Our implementation also fixes the problem described there.

In an asynchronous binary consensus protocol, every correct process initially *ac-proposes* a bit; the protocol concludes at a correct process when it *ac-decides* a bit. Our notion of Byzantine consensus uses strong validity in the asymmetric model. Furthermore, it restricts the safety properties of consensus from all correct ones to *wise* processes in the guild. For implementing asynchronous consensus, we use a system enriched with randomization. In the asymmetric model, the corresponding probabilistic termination property is guaranteed only for wise processes.

Definition 5.4 (Asymmetric strong Byzantine consensus)**.** *A protocol for asynchronous* asymmetric strong Byzantine consensus *satisfies:*

Probabilistic termination: *In all executions with a guild, every wise process* ac-decides *with probability* 1*, in the sense that*

$$\lim_{r \to +\infty} (\mathrm{P}[\textit{a wise process } p_i \textit{ ac-decides by round } r]) = 1.$$

Strong validity: *In all executions with a guild, a wise process only*

ac-decides a value that has been ac-proposed by some process in the maximal guild.

Integrity: *No correct process ac-decides twice.*

Agreement: *No two wise processes ac-decide differently.*

5.3.1 Asymmetric common coin

Our randomized consensus algorithm delegates its probabilistic choices to a *common coin* abstraction [6], [34]. This primitive is triggered by a *release-coin* invocation and terminates by generating an *output-coin*(s) event, where $s \in \mathcal{B}$ represents the random coin value in a range $\mathcal{B}$. We define this in the asymmetric-trust model.

Definition 5.5 (Asymmetric common coin)**.** *A protocol for* asymmetric common coin *satisfies the following properties:*

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

Unpredictability: *In all executions with a guild, no process has any information about the value of the coin before at least a kernel for all wise processes, which consists only of correct processes, has released the coin.*

Matching: *In all executions with a guild, with probability* 1 *every process in the maximal guild outputs the same coin value.*

No bias: *The distribution of the coin is uniform over* $\mathcal{B}$*.*

Here we consider binary consensus and $\mathcal{B} = \{0, 1\}$. The *termination* property guarantees that every process in the maximal guild eventually output a coin value that is ensured to be the same for each of them by the *matching* property. The *unpredictability* property ensures that the coin value is kept secret in an execution until at least a kernel for a wise process, consisting entirely of correct processes, releases the coin. The existence of a kernel made of correct processes is required in order to avoid the liveness problem at the consensus protocol presented in Section 5.2. The analogue of this in the threshold symmetric model, where $f < n/3$ processes may fail, would be a coin with threshold $2f$, where the value is kept secret until at least a set of $f+1$ *correct* processes release the coin. Finally, the *no bias* property specifies the probability distribution of the coin output.

The scheme. We recall here the notion of the *tolerated system* of an asymmetric Byzantine quorum system from Chapter 4. Every asymmetric Byzantine quorum system implies a tolerated system $\mathcal{T} = \{T_1, \ldots, T_K\}$, where $T_k = \mathcal{P} \setminus \mathcal{G}_k$ is the complement of guild $\mathcal{G}_k$, for each possible guild $\mathcal{G}_k$, where $k \in \{1, \ldots, K\}$. Crucial for our application is the fact that $\mathcal{T}$ satisfies the Q^3-condition (shown in Lemma 4.12), hence one can construct a *symmetric* Byzantine quorum system from $\mathcal{T}$: the canonical system $\mathcal{H} = \{\mathcal{G}_1, \ldots, \mathcal{G}_K\}$, consisting of all possible guilds $\mathcal{G}_k$, is such a symmetric Byzantine quorum system. The idea is to use the tolerated system as a "bridge" from the asymmetric to the symmetric model, since reasoning is simpler in the latter. In the same time, this approach guarantees that in any execution where the system is able to make progress (i.e., a guild exists), the processes in the guild will have enough information to reconstruct the value of the coin.

The common coin scheme follows the approach of Rabin [34] and assumes that coins are predistributed by a trusted dealer. The scheme uses Benaloh-Leichter [4] secret sharing, such that the coin is additively shared within every maximal guild. The dealer shares one coin for every possible round of the protocol. This requires knowledge of the symmetric Byzantine quorum system $\mathcal{H}$ corresponding to the tolerated system $\mathcal{T}$. Observe that every process can compute this because $\mathbb{F}$ is globally known.

We assume that before the coin protocol runs, the dealer has chosen uniformly at random a value $s \in \mathcal{B}$ and shared it as follows. For every possible guild $\mathcal{G} = \{p_{i_1}, \ldots, p_{i_m}\}$, the dealer has picked uniform shares $s_{i_1}, \ldots, s_{i_{m-1}}$ and set $s_{i_m} = s + \sum_{\ell=1}^{m-1} s_{i_\ell}$. Share s_{i_ℓ} has been given to process p_{i_ℓ}, for $\ell \in \{1, \ldots, m\}$. This implies that process p_i holds a share for every guild of which it is a member.

The code for process p_i to release the coin is shown in Algorithm 2. Specifically, when asked to release its coin share (Lines 5–9, Algorithm 2), a process p_i sends to all other processes a share $s_{\mathcal{G}}$ for each guild $\mathcal{G}$ of which p_i is a member. Upon receiving such shares, each process stores them in a local structure (Lines 11–13, Algorithm 2). When a process p_i has enough shares, i.e., all shares from a guild $\mathcal{G}$, it can locally add them and output the coin value (Lines 15–17, Algorithm 2).

Theorem 5.6. *Algorithm 2 implements an asymmetric common coin.*

Proof. Let us consider an asymmetric fail-prone system $\mathbb{F}$ such that $B^3(\mathbb{F})$ holds and the corresponding asymmetric Byzantine quorum system $\mathbb{Q}$ for $\mathbb{F}$. By Lemma 4.12, the tolerated system $\mathcal{T}$ of $\mathbb{Q}$ satisfies the

Algorithm 2 Asymmetric common coin for round *round* (code for p_i)

1: **State**
2: $\mathcal{H}$: set of all possible guilds
3: *share*$[\mathcal{G}][j]$: if $p_i \in \mathcal{G}$, this holds the share received from p_j
4: for guild $\mathcal{G}$; initially $\perp$

5: **upon event** *release-coin* **do**
6: **for all** $\mathcal{G} \in \mathcal{H}$ **such that** $p_i \in \mathcal{G}$ **do**
7: let $s_{i\mathcal{G}}$ be the share of p_i for guild $\mathcal{G}$
8: **for all** $p_j \in \mathcal{P}$ **do**
9: send message [SHARE, $s_{i\mathcal{G}}$, $\mathcal{G}$, *round*] to p_j

10: **upon** receiving a message [SHARE, s, $\mathcal{G}$, r] from p_j **such that**
11: $r =$ *round* **and** $p_j \in \mathcal{G}$ **do**
12: **if** *share*$[\mathcal{G}][j] = \perp$ **then**
13: *share*$[\mathcal{G}][j] \leftarrow s$

14: **upon exists** $\mathcal{G}$ **such that for all** j with $p_j \in \mathcal{G}$, it holds
15: *share*$[\mathcal{G}][j] \neq \perp$ **do**
16: $s \leftarrow \sum_{j:p_j \in \mathcal{G}}$ *share*$[\mathcal{G}][j]$
17: **output** *output-coin(s)*

Q^3-condition. Let $\mathcal{H}$ be the Byzantine quorum system for $\mathcal{T}$ consisting of all maximal guilds. Assume an execution with a guild, where all processes in some $T \in \mathcal{T}$ are faulty and $\mathcal{G} \in \mathcal{H}$ is the maximal guild.

For the *termination* property, observe that every correct process, and hence also every process in $\mathcal{G}$, invokes *release-coin*. This implies that eventually every process $p_i \in \mathcal{G}$ sends SHARE messages to all processes in $\mathcal{P}$ (Line 9, Algorithm 2), containing the coin shares of p_i for every guild in which p_i belongs (Line 6, Algorithm 2), including $\mathcal{G}$. Eventually every correct process in $\mathcal{P}$ receives a SHARE message from every process in $\mathcal{G}$, computes s (Line 16, Algorithm 2) and triggers *output-coin*(s).

For the *unpredictability* property, let us assume an execution with set of faulty processes F and with maximal guild $\mathcal{G} \in \mathcal{H}$. Moreover, let us assume that a correct process p_i has output coin s. This implies that there exists a set $\mathcal{G}_k \in \mathcal{H}$ such that every process in $\mathcal{G}_k$ has sent a SHARE message. From the consistency property of $\mathcal{H}$ the set $K = \mathcal{G}_k \setminus F$ intersects every quorum in $\mathcal{H}$.

We show that K is a kernel for all wise processes in such execution. Let us assume towards a contradiction that there exists a wise process

$p_j \in \mathcal{P}$ and a quorum $Q_j \in \mathcal{Q}_j$ for p_j such that $Q_j \cap K = \emptyset$. Observe that, by construction of K and by Lemma 4.7, K always contains a (wise) process in $\mathcal{G}$. This because K is derived by a possible guild $\mathcal{G}_k \in \mathcal{H}$ and, by definition of guild, every process in $\mathcal{G}_k$ has a quorum contained in $\mathcal{G}_k$. Seen that, by construction, $K = \mathcal{G}_k \setminus F$, then every process in K (which is correct) has a quorum in $\mathcal{G}_k$ containing at least a process in $\mathcal{G}$. Let p_i be a process in $K \cap \mathcal{G}$.

Since $K \subseteq \mathcal{G}_k$ and $\mathcal{G}_k$ is a possible guild in $\mathcal{H}$, there must exist a quorum $Q_i \in \mathcal{Q}_i$ for p_i such that $Q_i \subseteq \mathcal{G}_k$. Since $Q_j \cap K = \emptyset$ and $K = \mathcal{G}_k \setminus F$, it follows that $Q_i \cap Q_j \subseteq F$. This contradicts the consistency property of $\mathbb{Q}$, since both p_i and p_j are wise processes. This implies that K is a kernel for every wise process consisting of correct processes.

The *matching* and *no bias* properties follow directly from the fact that the coin value for every round is predetermined, albeit not known to any process, and chosen uniformly at random by the trusted dealer. □

Example 5.7. *Let us consider a five-process asymmetric quorum system $\mathbb{Q}_D$, defined through the following $\mathbb{F}_B$.*

$$\mathbb{F}_B: \begin{array}{rcl} \mathcal{F}_1 & = & \Theta_1^3(\{p_3, p_4, p_5\}) \\ \mathcal{F}_2 & = & \Theta_1^3(\{p_3, p_4, p_5\}) \\ \mathcal{F}_3 & = & \Theta_2^2(\{p_1, p_2\}) \vee \{p_4\} \vee \{p_5\} \\ \mathcal{F}_4 & = & \Theta_2^2(\{p_1, p_2\}) \vee \{p_3\} \vee \{p_5\} \\ \mathcal{F}_5 & = & \Theta_2^2(\{p_1, p_2\}) \vee \{p_3\} \vee \{p_4\} \end{array}$$

The tolerated system is $\mathcal{T} = \{\{p_1, p_2\}, \{p_3\}, \{p_4\}, \{p_5\}\}$. One can verify that $B^3(\mathbb{F}_B)$ holds; hence, by Lemma 4.12, also $Q^3(\mathcal{T})$ holds. The corresponding symmetric Byzantine quorum system is $\mathcal{H} = \{\{p_3, p_4, p_5\}, \{p_1, p_2, p_4, p_5\}, \{p_1, p_2, p_3, p_5\}, \{p_1, p_2, p_3, p_4\}\}$. Observe that every $\mathcal{G} \in \mathcal{H}$ a guild in an execution in which the processes in $T = \mathcal{P} \setminus \mathcal{G}$ are faulty.

Let us assume an execution with a set of faulty processes $F = \{p_1, p_2\}$; this implies that the guild in this execution is $\mathcal{G}_1 = \{p_3, p_4, p_5\}$.

Moreover, we consider just one round round=1 and we show how Algorithm 2 works. Let us assume that the dealer has chosen $s = 1$. Then, for every guild $\mathcal{G}_k \in \mathcal{H}$, with $k \in \{1, \ldots, 4\}$, the dealer has chosen uniform shares as follows, where s_i^k denotes the share of process i for guild $\mathcal{G}_k$.

$$\mathcal{G}_1 = \{p_3, p_4, p_5\} : s_3^1 = 1, s_4^1 = 0, \text{ and } s_5^1 = s + s_3^1 + s_4^1 = 0$$
$$\mathcal{G}_2 = \{p_1, p_2, p_4, p_5\} : s_1^2 = 0, s_2^2 = 1, s_4^2 = 1, \text{ and } s_5^2 = s + s_1^2 + s_2^2 + s_4^2 = 1$$
$$\mathcal{G}_3 = \{p_1, p_2, p_3, p_5\} : s_1^3 = 0, s_2^3 = 1, s_3^3 = 0, \text{ and } s_5^3 = s + s_1^3 + s_2^3 + s_3^3 = 0$$
$$\mathcal{G}_4 = \{p_1, p_2, p_3, p_4\} : s_1^4 = 1, s_2^4 = 0, s_3^4 = 0, \text{ and } s_4^4 = s + s_1^4 + s_2^4 + s_3^4 = 0$$

Every process in $\mathcal{G}_1 = \{p_3, p_4, p_5\}$ upon release-coin sends a SHARE *message to every process $p_j \in \mathcal{P}$ for every share it has.*

*Process p_3 is part of $\mathcal{G}_1, \mathcal{G}_3$, and $\mathcal{G}_4$. This means that upon release-coin, p_3 sends ⌈*SHARE*, 1, 1, 1⌋, ⌈*SHARE*, 0, 3, 1⌋, and ⌈*SHARE*, 0, 4, 1⌋ to every process in $\mathcal{P}$.*

*Process p_4 is part of $\mathcal{G}_1, \mathcal{G}_2$, and $\mathcal{G}_4$. This means that upon release-coin, p_4 sends ⌈*SHARE*, 0, 1, 1⌋, ⌈*SHARE*, 1, 2, 1⌋, and ⌈*SHARE*, 0, 4, 1⌋ to every process in $\mathcal{P}$.*

*Process p_5 is part of $\mathcal{G}_1, \mathcal{G}_2$, and $\mathcal{G}_3$. This means that upon release-coin, p_5 sends ⌈*SHARE*, 0, 1, 1⌋, ⌈*SHARE*, 1, 2, 1⌋, and ⌈*SHARE*, 0, 3, 1⌋ to every process in $\mathcal{P}$.*

Eventually every process in $\mathcal{G}_1$ receives a SHARE *message of the form ⌈*SHARE*, s_i^1, 1, 1⌋ from each process $p_i \in \mathcal{G}_1$, computes $s \leftarrow \sum_{i:p_i \in \mathcal{G}_1} s_i^1$ (Line 16, Algorithm 2) and output-coin(1).*

Discussion This implementation is expensive because the number of shares for one particular coin held by a process p_i is equal to the number of guilds in which p_i is contained. It would be more efficient to implement an asymmetric coin "from scratch" according to the protocols of Canetti and Rabin [12] or of Patra, Choudhury, and Rangan [32]. Alternatively, distributed cryptographic implementations are possible, for example, implementations relying on the hardness of the discrete logarithm problem [7].

5.3.2 Asymmetric binary validated broadcast

We generalize the binary validated broadcast as introduced by Mostéfaoui, Moumen, and Raynal [30] to the asymmetric-trust model. All safety properties are restricted to wise processes, and a guild is required for liveness. Every process may broadcast a binary value $b \in \{0, 1\}$ by invoking *abv-broadcast*(b). The broadcast primitive outputs at least one value b and possibly also both values through an *abv-deliver*(b) event, according to the following notion.

Definition 5.8 (Asymmetric binary validated broadcast)**.** *A protocol for* asymmetric binary validated broadcast *satisfies the following properties:*

Validity: *In all executions with a guild, let K be a kernel for every process in the maximal guild. If every process in K is correct and has* abv-broadcast *the same value $b \in \{0, 1\}$, then every wise process eventually* abv-delivers *b.*

Integrity: *In all executions with a guild, if a wise process* abv-delivers *some b, then b has been* abv-broadcast *by some process in the maximal guild.*

Agreement: *In all executions with a guild, if a wise process* abv-delivers *some value b, then every wise process eventually* abv-delivers *b.*

Termination: *In all executions with a guild, every wise process eventually* abv-delivers *some value.*

Note that it guarantees properties only for processes that are wise or even in the maximal guild. Liveness properties also assume there exists a guild.

Algorithm 3 works in the same way as the binary validated broadcast by Mostéfaoui, Moumen, and Raynal [30] (Section 5.2), but differs in the use of asymmetric quorums. The condition of receiving VALUE messages containing b from at least $f + 1$ processes is replaced by receiving such messages from a kernel K_i for process p_i. Furthermore, a quorum Q_i for p_i is needed instead of $2f + 1$ messages for abv-delivering a bit.

Theorem 5.9. *Algorithm 3 implements asymmetric binary validated broadcast.*

Proof. To prove the *validity* property, let us consider a kernel K for every process p_i in the maximal guild $\mathcal{G}_{\max}$. Moreover, let us assume that every process in K has *abv-broadcast* the same value $b \in \{0, 1\}$. Then, by definition of a kernel, K intersects every Q_i for every $p_i \in \mathcal{G}_{\max}$. According to the protocol, every process in $\mathcal{G}_{\max}$ eventually sends [VALUE,b] unless $sentvalue[b]$ = TRUE for some $p_i \in \mathcal{G}_{\max}$. However, if $sentvalue[b]$ = TRUE for p_i, process p_i has already sent [VALUE,b]. Since every process in the maximal guild eventually sends [VALUE,b], eventually every correct process p_j also receives [VALUE,b] from a kernel

Algorithm 3 Asymmetric binary validated broadcast (code for p_i)

```
1: State
2:       sentvalue ← [FALSE]^2: sentvalue[b] indicates whether p_i
3:                 has sent [VALUE, b]
4:       values ← [∅]^n: list of sets of received binary values
5:
6: upon event abv-broadcast(b) do
7:       sentvalue[b] ← TRUE
8:       send message [VALUE, b] to all p_j ∈ P
9:
10: upon receiving a message [VALUE, b] from p_j do
11:      if b ∉ values[j] then
12:           values[j] ← values[j] ∪ {b}
13:
14: upon exists b ∈ {0,1} such that
15:           {p_j ∈ P| b ∈ values[j]} ∈ K_i and
16:                ¬sentvalue[b] do                    // a kernel for p_i
17:      sentvalue[b] ← TRUE
18:      send message [VALUE, b] to all p_j ∈ P
19:
20: upon exists b ∈ {0,1} such that
21:           {p_j ∈ P| b ∈ values[j]} ∈ Q_i do       // a quorum for p_i
22:      output abv-deliver(b)
```

for itself (see Corollary 4.9) and sends [VALUE,b] unless $sentvalue[b] =$ TRUE. However, as above, if $sentvalue[b] =$ TRUE for p_j, process p_j has already sent [VALUE,b]. It follows that eventually every wise process receives a quorum for itself of values b and *abv-delivers* b.

For the *integrity* property, let us assume an execution with a maximal guild $\mathcal{G}_{\max}$. Suppose first that only Byzantine processes *abv-broadcast* b. Then, the set consisting of only these processes cannot form a kernel for any wise process. It follows that Line 16 of Algorithm 3 cannot be satisfied. If only naïve processes *abv-broadcast* b, then by the definition of a quorum system and by the assumed existence of a maximal guild, there is at least one quorum for every process in $\mathcal{G}_{\max}$ that does not contain any naïve processes (e.g., as in Example 4.8). All naïve processes together cannot be a kernel for processes in $\mathcal{G}_{\max}$. Again, Line 16 of Algorithm 3 cannot be satisfied. Finally, let us assume that a wise process p_i outside the maximal guild *abv-broadcasts* b. Then, p_i cannot be a kernel for every wise process: it is not part of the quorums inside

$\mathcal{G}_{\max}$. It follows that if a wise process *abv-delivers* some b, then b has been *abv-broadcast* by some processes in the maximal guild.

To show *agreement*, let F be the set of faulty processes and suppose that a wise process p_i has *abv-delivered* b. Then it has obtained [VALUE,b] messages from the processes in some quorum $Q_i \in \mathcal{Q}_i$ and before from a kernel $K = Q_i \setminus F$ for itself. Each correct process in K has sent [VALUE,b] message to all other processes. Consider any other wise process p_j. Since p_i and p_j are both wise, we have $F \in {\mathcal{F}_i}^*$ and $F \in {\mathcal{F}_j}^*$, which implies $F \in {\mathcal{F}_i}^* \cap {\mathcal{F}_j}^*$. It follows that K is also a kernel for p_j. Thus, p_j sends a [VALUE,b] message to every process. This implies that all wise processes eventually send [VALUE,b] to all processes. This also implies that eventually every process in $\mathcal{G}_{\max}$ sends [VALUE,b]. By Corollary 4.9, $\mathcal{G}_{\max}$ is a kernel for every correct process p_k. Thus, p_k sends a [VALUE,b] message to every process. Therefore eventually every wise process receives a quorum for itself of [VALUE,b] messages and *abv-deliver* b.

For the *termination* property, let us assume an execution with a maximal guild $\mathcal{G}_{\max}$ and set of faulty processes F. Note that in any execution, every process in $\mathcal{P} \setminus F$ *abv-broadcasts* some binary values. We show that there is a set $K \subseteq \mathcal{P} \setminus F$ such that K is a kernel for every process in the maximal guild consisting of correct processes and every process in K *abv-broadcasts* the same value $b \in \{0, 1\}$. Observe that a correct process initially *abv-broadcasts* only one value in $\{0, 1\}$. So, let $\mathcal{P} \setminus F = S_0 \cup S_1$ with S_0 and S_1 two sets of processes such that $S_0 \cap S_1 = \emptyset$ and such that every process in S_0 *abv-broadcasts* b and every process in S_1 *abv-broadcasts* $1 - b = \overline{b}$. Moreover, let us assume that neither S_0 nor S_1 contains a kernel for every process in the maximal guild. If S_0 does not contain a kernel for a process in the maximal guild, then there exists a process $p_j \in \mathcal{G}_{\max}$ and a quorum Q_j for p_j such that $Q_j \cap S_0 = \emptyset$. This means that every correct process in Q_j *abv-broadcasts* $\overline{b}$. Similarly, if S_1 does not contain a kernel for a process in the maximal guild, then there exists a process $p_k \in \mathcal{G}_{\max}$ and a quorum Q_k for p_k such that $Q_k \cap S_1 = \emptyset$. This means that every correct process in Q_k *abv-broadcasts* b. However, if this is the case, then $Q_j \cap Q_k \subseteq F$, which contradicts the consistency property of an asymmetric Byzantine quorum system, given that p_j and p_k are both wise. This implies that either S_0 or S_1 contains a kernel K for every process in the maximal guild consisting of correct processes and such that every process in K *abv-broadcasts* the same value. Termination then follows from the validity property. □

5.3.3 Asymmetric randomized consensus

In the following primitive, a correct process may *propose* a binary value b by invoking *ac-propose*(b); the consensus abstraction *decides* for b through an *ac-decide*(b) event.

Algorithms 4-5 differs from Algorithm 1 in some aspects. Recall that both algorithms use a system where messages are authenticated and delivered reliably. Importantly, we assume for Algorithms 4-5 that all messages among correct processes are also delivered in FIFO order, even when they do not originate from the same protocol module.

Moreover, Algorithms 4-5 allow the set B to change while reconstructing the common coin (Line 31). This step is necessary in order to prevent the problem described in Section 5.2. We prove this statement in Lemma 5.10.

Finally, our protocol may disseminate DECIDE messages in parallel to ensure termination. When p_i receives a DECIDE message from a kernel of processes for itself containing the same value b, it broadcasts a DECIDE message itself containing b to every processes, unless it has already done so. Once p_i receives a DECIDE message from a quorum of processes for itself with the same value b, it *ac-decides*(b) and halts. This "amplification" step is reminiscent of Bracha's reliable broadcast protocol [5]. Hence, the protocol does not execute rounds forever, in contrast to the original formulation of Mostéfaoui, Moumen, and Raynal [30] (Algorithm 1).

The following result shows that the problem described in Section 5.2 no longer occurs in our protocol.

Lemma 5.10. *If a wise process p_i outputs the coin with $B = \{0, 1\}$, then every other wise process that outputs the coin has also $B = \{0, 1\}$.*

Proof. Let us assume that a wise process p_i outputs the coin with $B = \{0, 1\}$. This means that p_i has received SHARE messages from a quorum Q_i^{COIN} for itself and has $B = aux[j] = \{0, 1\}$ for all $p_j \in Q_i$. Consider another wise process p_j that also outputs the coin. It follows that p_j has received SHARE messages from a quorum Q_j^{COIN} for itself as well. Observe that p_i and p_j, before receiving the SHARE messages from every process in Q_i^{COIN} and Q_j^{COIN}, respectively, receive all AUX messages that the correct processes in these quorums have sent before the COIN messages. This follows from the assumption of FIFO reliable point-to-point links. Quorum consistency implies that Q_i^{COIN} and Q_j^{COIN} have some correct processes in common. So, p_i and p_j receive some AUX messages

Algorithm 4 Asymmetric randomized binary consensus (code for p_i) Part 1

State
    $round \leftarrow 0$: current round
    $values \leftarrow \{\}$: set of *abv-delivered* binary values for the round
    $aux \leftarrow [\{\}]^n$: stores sets of values that have been received in
        AUX messages in the round
    $decided \leftarrow []^n$: stores binary values that have been reported
        as decided by other processes
    $sentdecide \leftarrow$FALSE: indicates whether p_i has sent a DECIDE message

upon event *ac-propose*(b) **do**
    invoke *abv-broadcast*(b) with tag *round*

upon *abv-deliver*(b) with tag r **such that** $r = round$ **do**
    $values \leftarrow values \cup \{b\}$
    send message [AUX, *round*, b] to all $p_j \in \mathcal{P}$

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

upon receiving a message [DECIDE, b] from p_j **such that**
        $decided[j] = \perp$ **do**
    $decided[j] = b$

upon exists $b \neq \perp$ **such that**
        $\{p_j \in \mathcal{P} \mid decided[j] = b\} \in \mathcal{K}_i$ **do** // a kernel for p_i
    if $\neg sentdecide$ **then**
        send message [DECIDE, b] to all $p_j \in \mathcal{P}$
        $sentdecide \leftarrow$TRUE

upon exists $b \neq \perp$ **such that**
        $\{p_j \in \mathcal{P} \mid decided[j] = b\} \in \mathcal{Q}_i$ **do** // a quorum for p_i
    ac-decide(b)
    halt // stop operating when aborted

upon exist $\{p_j \in \mathcal{P} \mid aux[j] \subseteq values\} \in \mathcal{Q}_i$ **do** // a quorum for p_i
    release-coin with tag *round*

Algorithm 5 Asymmetric randomized binary consensus (code for p_i) Part 2

30: **upon event** *output-coin*(s) with tag *round* **and**
31: **exits** $B \neq \{\}$ **such that** $\forall\, p_j \in Q_i, B = aux[j]$ **do**
32: $round \leftarrow round + 1$
33: **if exists** b **such that** $|B| = 1 \wedge B = \{b\}$ **then**
34: **if** $b = s \wedge \neg sentdecide$ **then**
35: send message [DECIDE, b] to all $p_j \in \mathcal{P}$
36: $sentdecide \leftarrow$TRUE
37: **invoke** *abv-broadcast*(b) with tag *round*
38: // propose b for the next round
39: **else**
40: **invoke** *abv-broadcast*(s) with tag *round*
41: // propose coin value s for the next round
42: $values \leftarrow [\perp]^n$
43: $aux \leftarrow [\{\}]^n$

from the same correct process before they may output the coin. This means that if p_i has $B = \{0, 1\}$ after the *output-coin* event, then every quorum Q_j for p_j will contain a process p_k such that $aux[k] = \{0, 1\}$ for p_j. Every wise process therefore must have $B = \{0, 1\}$ before it can proceed. □

The problem shown in Section 5.2 arose from messages between correct processes that were reordered in such a way that knowledge of the common coin value s was able to influence another correct process and cause it to deliver $\neg s$ alone. Lemma 5.10 above implies that our Algorithms 4-5 prevent this because all wise processes arrive at the same set B when they output the coin.

Theorem 5.11. *Algorithms 4-5 implement asymmetric strong Byzantine consensus.*

Proof. To prove the *strong validity* property, let us assume that a wise process p_i has *ac-decided* a value b. This means that p_i has received [DECIDE, b] messages from a quorum Q_i for itself. Moreover, before deciding, process p_i has received [DECIDE, b] messages from a kernel $\mathcal{K}_i$ for itself and sent [DECIDE, b] to every other process.

Whenever a correct process p_i has sent such a DECIDE message containing b in a round r, it has obtained $B = \{b\}$ and b is the same as the coin value in the round. Then, p_i has received b from a quorum Q_i for

itself through AUX messages. Every process in Q_i has received a [AUX, r, b] message and b has been *abv-delivered*. According to the integrity property of the validated broadcast, b has been *abv-broadcast* by a process in the maximal guild and, specifically, *values* contains only values *abv-broadcast* by processes in the maximal guild. It follows that b has been proposed by some processes in the maximal guild.

For the *agreement* property, suppose that a wise process has received [AUX, r, b] messages from a quorum Q_i for itself. Consider any other wise process p_j that has received a quorum Q_j for itself of [AUX, r, $\overline{b}$] messages. If at the end of round r there is only one value in B, then from consistency property of quorum systems, it follows $b = \overline{b}$. Furthermore, if $b = s$ then p_i and p_j broadcast a [DECIDE, b] message to every process and decide for b after receiving a quorum of [DECIDE, b] messages for themselves, otherwise they both *abv-broadcast*(b) and they continue to *abv-broadcast*(b) until $b = s$. If B contains more than one value, then p_i and p_j proceed to the next round and invoke a new instance of *abv-broadcast* with s. Therefore, at the beginning of the next round, the proposed values of all wise processes are equal. The property easily follows.

The *integrity* property is easily derived from the algorithm.

The *probabilistic termination* property follows from two observations. First, the termination and the agreement properties of binary validated broadcast imply that every wise process *abv-delivers* the same binary value from the validated broadcast instance and this value has been *abv-broadcast* by some processes in the maximal guild. Second, we show that with probability 1, there exists a round at the end of which all processes in $\mathcal{G}_{\max}$ have the same proposal b. If at the end of round r, every process in $\mathcal{G}_{\max}$ has proposed the coin value (Line 41, Algorithm 5), then all of them start the next round with the same value. Similarly, if every process in $\mathcal{G}_{\max}$ has executed Line 38 (Algorithm 5) they adopt the value b and start the next round with the same value.

However, it could be the case that some wise processes in the maximal guild proposes b and another one proposes the coin output s. Observe that the properties of the common coin abstraction guarantee that the coin value is random and independently chosen. So, the random value s is equal to the proposal value b with probability $\frac{1}{2}$. The probability that there exists a round r' in which the coin equals the value b proposed by all processes in $\mathcal{G}_{\max}$ during round r' approaches 1 when r goes to infinity.

Let r thus be some round in which every process in $\mathcal{G}_{\max}$ *abv-*

broadcasts the same value b; then, none of them will ever change their proposal again. This is due to the fact that every wise process invokes an binary validated broadcast instance with the same proposal b. According to the validity and agreement properties of asymmetric binary validated broadcast, every wise process then *bv-delivers* the same, unique value b. Hence, the proposal of every wise process is set to b and does not change in future rounds. Finally, the properties of common coin guarantee that the processes eventually reach a round in which the coin outputs b. Therefore, with probability 1 every process in the maximal guild sends a DECIDE message with value b to every process in that round. This implies that it exists a quorum $Q_i \subseteq \mathcal{G}_{\max}$ for a process $p_i \in \mathcal{G}_{\max}$ such that every process in Q_i has sent a DECIDE message with value b to every process. Moreover, the set of processes in the maximal guild is a kernel for p_i and for every other correct process p_j (Corollary 4.9). If a correct process p_j receives a DECIDE message with value b from a kernel for itself, it sends a DECIDE message with value b to every process unless it has already done so. It follows that eventually every wise process receives DECIDE messages with the value b from a quorum for itself and *ac-decides* for b. □

5.4 On asymmetric leader-based Byzantine consensus

In the consensus protocol presented in Section 5.3, no process had a different role within the protocol. This differs from consensus protocols such as PBFT [13], [14] or HotStuff [38], where, through the execution, a process behaves as a coordinator (or *leader*), and properties for such protocols are ensured with respect to its behavior. These protocols are generally referred to as *leader-based consensus protocols*. In particular, let us consider PBFT. This is a partially synchronous [16] consensus protocol that proceeds in *epochs*, identified with increasing timestamps, each of them with a designated leader, whose task is to reach consensus among the processes. If the leader is correct and no further epoch starts, then the leader succeeds in reaching consensus. But if the next epoch in the sequence is triggered, the processes abort the current epoch and invoke the next one, even if some processes may already have decided in the current epoch. In particular, to reach a consensus, two kind of operations are described: normal-case operation and epoch change. The normal-case operation occurs when there are no network delays or faulty

leader and it is made just by a Bracha's reliable broadcast [5] instance. If the system behaves correctly there is no need for epoch change operation. Otherwise, the operation is aborted, a new epoch starts and a new leader is elected. The epoch change part of the algorithm ensures *termination* by allowing the system to change leader in case it is Byzantine. When the leader is detected to be faulty, or there is a delay in the network communication, a new epoch starts with a new selected leader.

To better understand the tasks executed by the leader, we recall the abstraction used by Cachin, Guerraoui, and Rodrigues [6] called *conditional collect*, and we generalize it with a (symmetric) quorum system. The purpose of the conditional collect is to collect information in the system, in the form of messages from all processes, in a consistent way. This abstraction, which will be used within the consensus protocol, is parameterized by a predicate $C(\,)$, defined on a vector M of size n, i.e., the total number of processes in the system, of messages, and it should only output a collected vector that satisfies such predicate such that $M[i]$ is either equal to UNDEFINED or corresponds to the input message of process p_i. A conditional collect primitive should collect the same vector of messages at every correct process such that this vector satisfies the predicate. Correct processes must all input messages that will satisfy the predicate. More precisely, we say that the correct processes input *compliant* messages when each correct process p_i inputs a message m_i and any vector M with $M[i] = m_i$ satisfies $C(M)$.

Definition 5.12 (Quorum-based conditional collect)**.** *A protocol for* quorum-based conditional collect *satisfies:*

Consistency: *If the leader is correct, then every correct process collects the same M, and this M contains processes forming a quorum with messages different from* UNDEFINED*.*

Integrity: *If some correct process collects M with $M[j] \neq$* UNDEFINED *for some process p_i and p_i is correct, then p_i has input message $M[i]$.*

Termination: *If all correct processes input compliant messages and the leader is correct, then every correct process eventually collects some M such that $C(M) =$* TRUE*.*

It can be shown [6] that Algorithm 6 implements quorum-based conditional collect.

Algorithm 6 Quorum-based signed conditional collect *cc* with leader p_ℓ (Code for p_i)

```
1: State
2:     messages ← [UNDEFINED]^n: set of messages sent by processes
3:     Σ ← [⊥]^n: signatures of the processes
4:     collected ← FALSE

5: upon event input(m) do
6:     σ ← sign(p_i, cc ‖ p_i ‖ INPUT ‖ m)
7:     send message [SEND, m, σ] to leader p_ℓ

8: upon receiving a message [SEND, m, σ] from p_j do      // only leader p_ℓ
9:     if verifysig(p_j, cc ‖ p_j ‖ PARALLEL ‖ m, σ) then
10:        messages[j] ← m
11:        Σ[j] ← σ

12: upon exists m ≠ UNDEFINED such that
13:        {p_j ∈ P| messages[j] = m} ∈ Q and
14:            C(messages) do                              // only leader p_ℓ
15:    send message [COLLECTED, messages, Σ] to all p_j ∈ P
16:    messages ← [UNDEFINED]^n
17:    Σ ← [⊥]^n

18: upon receiving a message [COLLECTED, M, Σ] from p_ℓ do
19:    if ¬collected and exists m ≠ UNDEFINED such that
20:            {p_j ∈ P| M[j] = m} ∈ Q and C(M) and
21:                for all p_j ∈ P such that M[j] ≠ UNDEFINED it holds
22:                    verifysig(p_j, cc ‖ p_j ‖ INPUT ‖ M[j], Σ[j]) then
23:        collected ← TRUE
24:        output collected(M)
```

Cachin, Guerraoui, and Rodrigues [6] analyze PBFT [13] in a modular way, deconstructing it into different modules. In Algorithms 7-8 we present, and generalize, the consensus module of PBFT, called here *quorum-based Byzantine read/write epoch consensus*, which makes use of the conditional collect implemented in Algorithm 6.

Algorithms 7-8 can be seen as divided in a *read* phase (Algorithm 7) and a *write* phase (Algorithm 8). The read phase collects states from all processes to ascertain if a value might have already *decided* in a previous epoch. In the presence of Byzantine processes, the leader might write an wrong value. For this reason each process must replicate the

leader's computation and write a value to verify the leader's choice. The algorithm starts by the leader sending a READ message to all processes, which triggers every process to invoke a conditional collect primitive (Algorithm 6). Every process inputs a message [STATE, *valts*, *val*, *writeset*] containing its state. The leader in conditional collect is the leader p_ℓ of the epoch. In the write phase, when a process has received a quorum $Q \in \mathcal{Q}$ of WRITE messages containing the same value v, it sets its state to (ets, v) and broadcasts an ACCEPT message with v. When a process has received a quorum $Q \in \mathcal{Q}$ of ACCEPT messages containing the same value v, it *decides* v.

In Algorithms 7-8 we make use of a conditional collect with the aim of collecting information in the system in the form of messages from all processes in a way that is consistent among them.

Every correct process in Algorithms 7-8 initializes the conditional collect with a predicate $sound(\cdot)$ defined as

$$sound(states) \equiv (\exists\ (ts, v) \mid binds(ts, v, states)) \vee unbound(states))$$

which determines whether there exists a value that must be written during the write phase. If such a value exists, the read phase must identify it or conclude that no such value exists.

We say that *states binds* ts to v, writing $binds(ts, v, states)$, whenever

$$\{p_j \in \mathcal{P} |\ states[j] = [\text{STATE}, ts, v, ws]\} \in \mathcal{Q}$$

for some ts, v, ws and *both* the following conditions hold.

1. given a timestamp/value pair (ts, v),

$$\{p_j \in \mathcal{P} |\ states[j] = [\text{STATE}, ts', v', \cdot] \wedge \\ ts' < ts \vee (ts', v') = (ts, v)\} \in \mathcal{Q}$$

2. given a timestamp/value pair (ts, v), there exists ws' such that

$$\{p_j \in \mathcal{P} |\ states[p] = [\text{STATE}, \cdot, \cdot, ws'] \wedge \\ \exists\ (ts', v') \in ws' \text{ such that } ts' \geq ts \wedge v' = v\} \in \mathcal{C}$$

with $\mathcal{C}$ a set of *core sets* [21], [22]; a core set is a minimal set of processes that contains at least one correct process in every execution.

Algorithm 7 Quorum-based Byzantine Read/Write Epoch Consensus (Code for p_i) Read phase

1: **State**
2: *valts*: timestamp of the current state of a process
3: *val*: value of the current state of a process
4: *writeset*: timestamp/value pairs of the current state
5: that a process has ever written
6: $states[i] \leftarrow (valts, val, writeset)$
7: $written \leftarrow [\perp]^n$
8: $accepted \leftarrow [\perp]^n$
9: *ets*: instance of the epoch consensus with timestamp *ts*
10: *cc*: instance of the conditional collect primitive

11: **upon input** *propose*(v) **do** // only leader p_ℓ
12: **if** $val = \perp$ **then** $val \leftarrow v$
13: send message [READ] to all $p_j \in \mathcal{P}$

14: **upon** receiving a message [READ] from p_ℓ
15: input a message [STATE, *valts*, *val*, *writeset*] to *cc*

16: **upon output** *collected*(*states*) **do**
17: // $states[i]$ = [STATE, *ts*, v, *ws*] or $states[i]$ = UNDEFINED
18: $tmpval \leftarrow \perp$
19: **if exists** $ts \geqslant 0, v \neq \perp$ from *states* **such that**
20: *binds*($ts, v, states$) **then**
21: $tmpval \leftarrow v$
22: **else if exists** $v \neq \perp$ **such that**
23: *unbound*(*states*) **and** $states[\ell]$ = [STATE, $\cdot$, v, $\cdot$] **then**
24: $tmpval \leftarrow v$
25: **if** $tmpval \neq \perp$ **then**
26: **if exists** *ts* **such that** $(ts, tmpval) \in writeset$ **then**
27: $writeset \leftarrow writeset \setminus \{(ts, tmpval)\}$
28: $writeset \leftarrow writeset \cup \{(ets, tmpval)\}$
29: send message [WRITE, *tmpval*] to all $p_j \in \mathcal{P}$

We say that *states* is *unbound*, writing *unbound*(*states*), whenever

$$\{p_j \in \mathcal{P} |\ states[j] = [\text{STATE}, ts, v, ws]\} \in \mathcal{Q}$$

for some ts, v, ws and the timestamps of the entries in *states* of all the processes in some quorum are equal to 0.

Although this formalization works in a system where every process

Algorithm 8 Quorum-based Byzantine Read/Write Epoch Consensus (Code for p_i) Write phase

30: **upon** receiving a message [WRITE, *tmpval*] from p_j **do**
31: $written[j] \leftarrow v$

32: **upon exists** v **such that** $\{p_j \in \mathcal{P} |\ written[j] = v\} \in \mathcal{Q}$ **do**
33: $(valts, val) \leftarrow (ets, v)$
34: $written \leftarrow [\perp]^n$
35: send message [ACCEPT, *val*] to all $p_j \in \mathcal{P}$

36: **upon** receiving a message [ACCEPT, v] from p_j
37: $accepted[j] \leftarrow v$

38: **upon exists** v **such that** $\{p_j \in \mathcal{P} |\ accepted[j] = v\} \in \mathcal{Q}$ **do**
39: $accepted \leftarrow [\perp]^n$
40: **output** *decide*(v)

41: **upon input** *abort* **do**
42: **output** *aborted*(*valts*, *val*, *writeset*)
43: **halt** // stop operating when aborted

has the same quorum system $\mathcal{Q}$, we show subsequently that such a formulation faces certain issues within an asymmetric-trust model

Problem with asymmetric trust Algorithms 6-7-8 show that the leader, upon collecting a quorum Q of messages, executes some tasks, and broadcasts the result of these tasks (together with the received quorum Q of processes that sent the messages, as a certificate) to every other processes. At that point, every process, upon receiving the result and the certificate from the leader, repeats the same tasks that the leader did, in order to verify that it did not behave maliciously. It can be shown [6] that, if the leader is correct, this verification will output the same result as the leader's.

A key observation here is that both the leader and the other processes verify the messages based on a quorum Q, which is the same for every process. However, if we consider an asymmetric quorum system, where every process p_i has its own quorum system $\mathcal{Q}_i$, then $\mathcal{Q}_i$ might not contain any quorum Q_i such that $Q_i = Q_\ell$, with Q_ℓ a quorum for the leader p_ℓ. In other terms, if a leader p_ℓ receives messages from a quorum Q_ℓ for itself, it executes some tasks based on Q_ℓ, and it broadcasts the

result of these tasks together with the certificate based on Q_ℓ to every process, then a process p_i, upon receiving what p_ℓ sent, should verify based on its own quorum system $\mathcal{Q}_i$, possibly outputting a different result.

Because of that, one cannot just substitute a symmetric quorum system $\mathcal{Q}$ with an asymmetric quorum system $\mathbb{Q}$ without doing some required considerations and changes.

5.4.1 A solution to the problem

We present a possible solution to the problem presented above by means of the tolerated system of an asymmetric quorum system, as discussed in Section 4.2.3.

Recall that every asymmetric Byzantine quorum system (where at least a guild can be obtained) implies a tolerated system $\mathcal{T} = \{T_1, \ldots, T_K\}$, where $T_k = \mathcal{P} \setminus \mathcal{G}_k$ is the complement of guild $\mathcal{G}_k$, for each possible guild $\mathcal{G}_k$, where $k \in \{1, \ldots, K\}$. One can construct a symmetric Byzantine quorum system from $\mathcal{T}$: the canonical system $\mathcal{H} = \{\mathcal{G}_1, \ldots, \mathcal{G}_K\}$, consisting of all possible guilds $\mathcal{G}_k$, is such a symmetric Byzantine quorum system, connecting the asymmetric to the symmetric model.

So, let $\mathcal{T} = \{T_1, \ldots, T_K\}$ be the tolerated system for $\mathbb{Q}$, and let $\mathcal{H} = \{\mathcal{G}_1, \ldots, \mathcal{G}_K\}$ be a symmetric quorum system consisting of all possible guilds $\mathcal{G}_k$. We modify the quorum-based conditional collect primitive from Algorithm 6 in order to obtain Algorithm 9.

Algorithm 9 differs from Algorithm 6 in Line 13 and Line 20. In particular, since $\mathbb{Q}$ is known to every process in the system, so are the tolerated system $\mathcal{T}$ and the derived symmetric quorum system $\mathcal{H}$. In other terms, we parameterize the conditional collect through the symmetric Byzantine quorum system originated from the asymmetric trust of the system. In this way, given an execution, every process can verify the output of the leader through $\mathcal{H}$.

Algorithm 9 is parameterized also by a predicate C which, in the context of the quorum-based Byzantine read/write epoch consensus protocol (Algorithm 7-8) corresponds to

$$\mathit{sound}(\mathit{states}) \equiv (\exists\ (ts, v) \mid \mathit{binds}(ts, v, \mathit{states})) \vee \mathit{unbound}(\mathit{states}))\,.$$

Since *sound* is also parameterized by a symmetric Byzantine quorum system through *bind* and *unbound*, we require for *sound*, in order to

Algorithm 9 Asymmetric signed conditional collect *cc* with leader p_ℓ (Code for p_i)

```
1: State
2:     messages ← [UNDEFINED]^n: set of messages sent by processes
3:     Σ ← [⊥]^n: signatures of the processes
4:     collected ← FALSE

5: upon event input(m) do
6:     σ ← sign(p_i, cc ‖ p_i ‖ INPUT ‖ m)
7:     send message [SEND, m, σ] to leader p_ℓ

8: upon receiving a message [SEND, m, σ] from p_j do      // only leader p_ℓ
9:     if verifysig(p_j, cc ‖ p_j ‖ PARALLEL ‖ m, σ) then
10:        messages[j] ← m
11:        Σ[j] ← σ

12: upon exists m ≠ UNDEFINED such that
13:        {p_j ∈ P| messages[j] = m} ∈ H and
14:            C(messages) do                              // only leader p_ℓ
15:    send message [COLLECTED, messages, Σ] to all p_j ∈ P
16:    messages ← [UNDEFINED]^n
17:    Σ ← [⊥]^n

18: upon receiving a message [COLLECTED, M, Σ] from p_ℓ do
19:    if ¬collected and exists m ≠ UNDEFINED such that
20:            {p_j ∈ P| M[j] = m} ∈ H and C(M) and
21:                for all p_j ∈ P such that M[j] ≠ UNDEFINED it holds
22:                    verifysig(p_j, cc ‖ p_j ‖ INPUT ‖ M[j], Σ[j]) then
23:        collected ← TRUE
24:        output collected(M)
```

work in an asymmetric-trust model, to consider the Byzantine quorum system $\mathcal{H}$. Properties of Algorithm 9 directly follows from those of Algorithm 6, since we are considering a symmetric Byzantine quorum system $\mathcal{H}$.

At this point, the remaining part of the quorum-based Byzantine read/write epoch consensus can be generalized as it follows. Observe that the read phase (Algorithm 7) does not make use of quorums (except for the conditional collect, which we already generalized), so for simplicity we only generalize the write phase (Algorithm 8).

Algorithm 10 Asymmetric Byzantine Read/Write Epoch Consensus (Code for p_i) Write phase.

25: **upon** receiving a message [WRITE, *tmpval*] from p_j **do**
26: $written[j] \leftarrow v$

27: **upon exists** v **such that** $\{p_j \in \mathcal{P} |\ written[j] = v\} \in \mathcal{Q}_i$ **do**
28: $(valts, val) \leftarrow (ets, v)$
29: $written \leftarrow [\perp]^n$
30: send message [ACCEPT, *val*] to all $p_j \in \mathcal{P}$

31: **upon** receiving a message [ACCEPT, v] from p_j
32: $accepted[j] \leftarrow v$

33: **upon exists** v **such that** $\{p_j \in \mathcal{P} |\ accepted[j] = v\} \in \mathcal{Q}_i$ **do**
34: $accepted \leftarrow [\perp]^n$
35: **output** *decide*(v)

36: **upon input** *abort* **do**
37: **output** *aborted*($valts, val, writeset$)
38: **halt** // stop operating when aborted

Theorem 5.13. *Algorithms 7-10, implemented with the asymmetric signed conditional collect (Algorithm 9), satisfy the following properties. In all executions with a guild,*

Validity: *If a wise process decides v, then v was proposed by the leader p_ℓ of some epoch consensus with timestamp ts' $\leq$ ts and leader $p_{\ell'}$.*

Agreement: *No two correct processes decide differently.*

Integrity: *Every correct process decides at most once.*

Lock-in: *If a wise process has decided v in an epoch consensus with timestamp ts' $\leq$ ts, then no correct process decides a value different from v.*

Termination: *If the leader p_ℓ is correct, has proposed a value, and no wise process aborts this epoch consensus, then every process in $\mathcal{G}_{max}$ eventually decides some value.*

Abort Behavior: *When a correct process aborts an epoch consensus, it eventually will have completed the abort; moreover, a correct*

process completes an abort only if the epoch consensus has been aborted by some correct process.

Proof. For the *validity* property, let us assume that a wise process p_i *decides* for a value v. Process p_i only *decides* for the value v received in an ACCEPT message from a quorum for itself and that any correct process only sends an ACCEPT message with v after receiving v in a WRITE message from a quorum for itself. Any correct process only sends a WRITE message with v either after collecting a vector *states* that *binds ts* to v or after collecting *states* that is *unbound* and choosing v as proposed by the leader p_ℓ. For the latter, the validity property easily follows. For the former, one can apply the same reasoning backward, until it reaches an epoch where *states* is *unbound*. Validity property follows.

The *agreement* property follows from the consistency property of an asymmetric Byzantine quorum system, while the *integrity* property can be seen from the protocol.

The reasoning for the proof of the *lock-in* property closely mirrors the approach adopted in the proof presented by Cachin, Guerraoui, and Rodrigues in Section 5.6.3 of [6], although with a change in the quorum structure. Specifically, we are considering an asymmetric Byzantine quorum system in our case.

For the *termination* property, given that the asymmetric conditional collect as implemented in Algorithms 7-10 makes use of the symmetric Byzantine quorum system derived from the tolerated system, one can apply the same reasoning as Cachin, Guerraoui, and Rodrigues in Section 5.6.3 of [6]. However, it should be noted that once all the correct processes have verified the computation performed by the leader and sent a WRITE message with value v, every wise process p_i eventually receives a quorum Q_i for itself of such messages and sends an ACCEPT message. Eventually, every process p_j in the maximal guild receives a quorum Q_j for itself of ACCEPT messages with value v and *decides* for v. The termination property thus follows.

The *abort behavior* property is satisfied because the algorithm returns an event *aborted*(*states*) immediately and only if it has been aborted. □

5.4.2 Future improvements

The proposed solution, although it solves the issue described at the beginning of this section, comes with some trade-off. In particular, it is clear that we need to rely on a symmetric Byzantine quorum system in

order to verify the work of the leader, losing part of the subjectivity of the asymmetric-trust model. However, observe that such a symmetric quorum system is actually derived from an asymmetric system; in other words, it can be seen as a good compromise, in order to achieve consensus in an asymmetric setting.

A possible improvement to this solution could be to devise a composition rule among the quorum systems in an asymmetric Byzantine quorum system, in order to correctly verify the tasks done by the leader. This composition rule should take into account wise processes, in a given execution with a guild. In particular it should be in a way that, given the asymmetric conditional collect parameterized with asymmetric quorums, if the leader p_ℓ is wise, then every other process p_i in the guild collects the same M, and this M contains processes forming a quorum Q_i for p_i with messages different from UNDEFINED, end it should ensure that if all the wise processes input compliant messages and the leader is wise, then every process in the guild eventually collect some M such that $C(M) = \text{TRUE}$. Another improvement, more into the direction of HotStuff [38], would be to better understand how threshold signature schemes generalizes to the asymmetric-trust model. In this way, one could devise HotStuff allowing for signature verification based on asymmetric quorums. We leave these open questions for future works.

Chapter 6

Asymmetric Trust in Permissionless Networks

Fail-prone systems as studied so far require every process to know the full system membership in order to guarantee safety through globally intersecting quorums. Thus, they are of little help in an open, permissionless setting, where such knowledge may not be available. In this chapter, we propose a model that expands the theory of fail-prone systems to make it applicable to permissionless systems. Our model generalizes existing models such as the symmetric fail-prone system model [26] and the asymmetric fail-prone system model [9]. Moreover, it gives a characterization with standard formalism of the model used by the Stellar blockchain. The content of this chapter is taken from the paper "Quorum systems in permissionless networks" [8].

6.1 System model

Processes. Differently from Chapter 4 and Chapter 5, here we consider an *unbounded* set of processes $\mathcal{P} = \{p_1, p_2, ...\}$ that communicate asynchronously with each other by sending messages. Moreover, we assume that processes do not necessarily know which other processes are in the system (i.e., each process only knows a subset of $\mathcal{P}$).

Links. We assume that point-to-point communication between any two processes (that know each other) is available, as well as a best-effort

gossip primitive that will reach all processes.

6.2 Preliminaries

Processes make failure assumptions about other processes. However, since a process does not know exactly who is part of the system, it cannot make failure assumptions about the whole system. Instead, each process p_i makes assumptions about a set $P_i \subseteq \mathcal{P}$, called p_i's *trusted set*, using a *symmetric fail-prone system* $\mathcal{F}_i$ over P_i. We say that P_i and $\mathcal{F}_i$ constitute p_i's *assumptions* and they remain fixed during an execution.

A *permissionless fail-prone system* (abbreviated PFPS) describes the assumptions of all the processes:

Definition 6.1 (Permissionless fail-prone system)**.** *A* permissionless fail-prone system *is an array* $\mathbb{F} = [(P_1, \mathcal{F}_1), (P_2, \mathcal{F}_2), \ldots]$ *that associates each process* p_i *to a trusted set* $P_i \subseteq \mathcal{P}$ *and a fail-prone system* $\mathcal{F}_i$ *over* P_i*. We refer to* $(P_i, \mathcal{F}_i)$ *as the* configuration of process p_i*.*

We now consider a fixed PFPS $\mathbb{F}$.

Definition 6.2 (Tolerated execution and tolerated set)**.** *We say that* the assumptions of a process p_i are satisfied *in an execution if the set* A *of processes that actually fail is such that there exists a fail-prone set* $F \in \mathcal{F}_i$ *and:*

1. $A \cap P_i \subseteq F$*; and*
2. *the assumptions of every member of* $P_i \setminus F$ *are satisfied.*

If $p_i \in \mathcal{P}$ *has its assumptions satisfied in an execution* e*, we say that* p_i tolerates the execution e*.*

Finally, a set of processes L tolerates a set *of processes* A *if and only if every process* $p_i \in L \setminus A$ *tolerates an execution* e *with set of faulty processes* A*.*

Example 6.3. *Consider a set of processes* $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$ *and a permissionless fail-prone system* $\mathbb{F} = [(\mathcal{P}, \mathcal{F}_1), (\mathcal{P}, \mathcal{F}_2), (\mathcal{P}, \mathcal{F}_3), (\mathcal{P}, \mathcal{F}_4)]$ *with* $\mathcal{F}_1 = \{\{p_3, p_4\}\}$*,* $\mathcal{F}_2 = \{\{p_1, p_4\}\}$*,* $\mathcal{F}_3 = \{\{p_1, p_4\}\}$*, and* $\mathcal{F}_4 = \{\{p_1, p_2\}\}$*. Then,* $\mathcal{P}$ *tolerates the sets* $\emptyset$*,* $\{p_1\}, \{p_4\}$ *and* $\{p_1, p_4\}$*. To see this, let us assume an execution* e *with set of faulty processes* $A = \{p_1, p_4\}$*. Then, for every* $p_i \in \mathcal{P} \setminus A$*, there exists a fail-prone set* $F \in \mathcal{F}_i$ *such that* $A \cap \mathcal{P} \subseteq F$*. In particular,* $\mathcal{P} \setminus A = \{p_2, p_3\}$ *and* $\{p_1, p_4\} \in \mathcal{F}_2$ *and* $\{p_1, p_4\} \in \mathcal{F}_3$*. The same reasoning can be applied for the other sets.*

Note that here we depart significantly from the traditional notion of fail-prone systems [9], [26]: in a PFPS, a process not only makes assumptions about failures, but also makes assumptions about the assumptions of other processes.

Next we define *survivor sets* analogously to Junqueira and Marzullo [21]. In the traditional literature, a survivor set of p_i is the complement, within $\mathcal{P}$, of some fail-prone set. However, defining them as the complement of fail-prone sets within P_i does not work because of Item 2 in Definition 6.2. To obtain this definition, we first define a *slice*.

Definition 6.4 (Slice)**.** *A set $\overline{F} \subseteq \mathcal{P}$ is a* slice *of p_i if and only if p_i has a fail-prone set $F \in \mathcal{F}_i$ such that $\overline{F} = P_i \setminus F$.*

For any $S \subseteq \mathcal{P}$ we often say *p_i has a slice in S* when a slice of p_i is contained in S or when S contains a superset of a slice of p_i.

Definition 6.5 (Survivor-set system)**.** *A* survivor-set system *$\mathcal{S}_i$ of p_i is the minimal set of subsets S of $\mathcal{P}$ such that:*

1. *p_i has a slice in S; and*

2. *every member of S has a slice in S.*

Each $S \in \mathcal{S}_i$ is called a survivor set *of p_i.*

Example 6.6. *Continuing from Example 6.3, process p_1 has only one slice consisting of $\{p_1, p_2\}$, processes p_2 and p_3 have the set $\{p_2, p_3\}$ as slice, and process p_4 has the set $\{p_3, p_4\}$ as slice. Moreover, the survivor-set systems are $\mathcal{S}_1 = \{\{p_1, p_2, p_3\}, \{p_1, p_2, p_3, p_4\}\}$ for process p_1, $\mathcal{S}_2 = \{\{p_2, p_3\}, \{p_1, p_2, p_3, p_4\}\}$ for process p_2, $\mathcal{S}_3 = \{\{p_2, p_3\}, \{p_1, p_2, p_3, p_4\}\}$ for process $p3$, and $\mathcal{S}_4 = \{\{p_2, p_3, p_4\}, \{p_1, p_2, p_3, p_4\}\}$ for process p_4. This follows from Definition 6.5: given a survivor set $S \in \mathcal{S}_i$ for p_i, process p_i must have a slice in S and every member of S must have a slice in S. So, for example, given the survivor set $\{p_1, p_2, p_3\}$ in the survivor set system $\mathcal{S}_1$ for p_1, process p_1 has a slice in $\{p_1, p_2, p_3\}$, i.e., $\{p_1, p_2\}$, and every process $p_i \in \{p_1, p_2, p_3\}$ has a slice in $\{p_1, p_2, p_3\}$, i.e., $\{p_2, p_3\}$.*

Lemma 6.7. *The assumptions of a process $p_i \in \mathcal{P}$ are satisfied in an execution e with set of faulty processes A if and only if there exists a survivor set $S \in \mathcal{S}_i$ of p_i such that S does not fail.*

Proof. Let p_i be a process such that, given an execution e with set of faulty processes A, the assumptions of p_i are satisfied in e. This implies that, by Definition 6.2, there exists a set of processes such that each of these processes has its assumptions satisfied. Moreover, by Definition 6.4, each of these processes has a slice $\overline{F}_j$ such that $\overline{F}_j \cap A = \emptyset$. This leads to have a set S obtained as union of all of these slices such that $S \cap A = \emptyset$ and such that S is minimal with respect to this union, in the sense that is the minimal set of processes such that every process in S has its assumptions satisfied. The set S is a survivor set of p_i.

Conversely, we show that given a survivor set S of p_i, given a process $p_i \in S$ and given an execution e with set of faulty processes A, if $S \cap A = \emptyset$, then the assumptions of p_i are satisfied in e. Observe that, from the assumptions, we have that every process in S has a slice $\overline{F}$ in S such that $\overline{F} \cap A = \emptyset$. This means that for every process p_i in S, there exists a fail-prone set $F \in \mathcal{F}_i$ such that $P_i \cap A \subseteq F$. This implies that every process in S has its assumptions satisfied and, in particular, that $p_i \in S$ has its assumptions satisfied in e. □

6.3 Permissionless Byzantine quorum systems

A symmetric fail-prone system [26] determines a canonical Byzantine quorum system known to all processes through the Q^3-condition. Specifically, given a fail-prone system $\mathcal{F}$, the Q^3-condition requires that no three fail-prone sets of $\mathcal{F}$ cover the complete set of processes and this condition holds if and only if there exists a quorum system for $\mathcal{F}$ [20], [26]. Such a quorum system could be, for example, the complement of every fail-prone set of $\mathcal{F}$, which we call the *canonical* quorum system. Traditional algorithms such as read-write register emulations [26], Byzantine reliable broadcasts [5], [35] or the PBFT algorithm [14] make use of quorums.

In the model of asymmetric trust [15] the assumptions of the processes may differ, and asymmetric Byzantine quorum systems [9], [10] allow to implement the above-mentioned algorithms in a more flexible way. However, they still require a system that is known to every process.

In a permissionless system, processes do not know the membership and have different, partial, and potentially changing views of its composition.

Given a PFPS, we would therefore like to obtain a quorum system to

implement algorithms for register emulation, broadcast, consensus and more, while allowing the processes to have different assumptions in an open network.

We are therefore interested in defining a notion of quorums for open systems where:

1. each process has its own quorum system; and
2. the quorums of a process p_i depend on the assumptions of other processes, which p_i learns by communicating with them.

In other words, we consider scenarios in which each process p_i communicates with other processes, continuously discovers new processes, and learns their assumptions. During this execution, p_i determines its current set of quorums as a function of what it has learned so far. Importantly, this means that the quorums of a process evolve as the process learns new assumptions, and that faulty processes can influence p_i's quorums by lying about their assumption.

We now formalize this model using the notions of a *view* and a *quorum function*.

Definition 6.8 (View)**.** *A* view $\mathbb{V} = [\mathcal{V}_1, \mathcal{V}_2, \ldots]$ *is an array with one entry* $\mathbb{V}[j] = \mathcal{V}_j$ *for each process* p_j *such that:*

1. *either* $\mathcal{V}_j$ *is the special value* $\perp$*; or*
2. $\mathcal{V}_j = (P_j, \mathcal{F}_j)$ *consists of a set of processes* P_j *and a fail-prone system* $\mathcal{F}_j$ *over* P_j*.*

Observe that every process p_i has its *local* view $\mathbb{V}$, whose non-$\perp$ entries represent the assumptions that p_i has learned at some point in an execution. Every other process p_j such that $\mathbb{V}[j] = \perp$ is a process that p_i has not heard from. We denote with Υ the set of all the possible views.

We assume that, for every process p_j, a process p_i's view contains the assumption that p_i has most recently received from p_j. Finally, note that $\mathbb{F}$ is a view in which no process is mapped to $\perp$. In particular, $\mathbb{F}$ represents the global view if the system could be entirely observed. Since processes cannot observe the complete system, they normally only have partial knowledge of $\mathbb{F}$. Moreover, this knowledge evolves over time.

Definition 6.9 (Domain of a view)**.** *For a view* $\mathbb{V}$*, the set of processes* p_i *such that* $\mathbb{V}[i] \neq \perp$ *is the* domain *of* $\mathbb{V}$*.*

Next, we assume that every process determines its quorums according to its view using a function $\mathscr{Q}$ called a *quorum function*. We assume that all correct processes use the same $\mathscr{Q}$ and that they do not change it during an execution. We then have the following definition.

Definition 6.10 (Quorum function)**.** *The* quorum function $\mathscr{Q} : \mathcal{P} \times \Upsilon \to 2^{\mathcal{P}}$ *maps a process* p_i *and a view* $\mathbb{V}$ *to a set of sets of processes such that* $Q \in \mathscr{Q}(p_i, \mathbb{V})$ *if and only if:*

1. *a slice of* p_i *is contained in* Q*; and*
2. *for every process* $p_j \in Q$ *with* $\mathbb{V}[j] \neq \perp$ *and* $\mathbb{V}[j] = (P_j, \mathcal{F}_j)$*, there exists* $F \in \mathcal{F}_j$ *such that* $P_j \setminus F \subseteq Q$*.*

Every element of $\mathscr{Q}(p_i, \mathbb{V})$ *is called a* permissionless quorum for p_i *(in* $\mathbb{V}$*).*

In the context of this chapter, whenever it is clear from the context, we refer to a permissionless quorum simply as *quorum*. Notice that in the first condition, the quorum Q may itself be a slice of p_i. Moreover, Q is a quorum for every one of its members and it is defined by slices of every $p_i \in Q$. As shown in the following lemma, a quorum for p_i in view $\mathbb{V}$ for p_i is a survivor set of p_i.

Lemma 6.11. *For every view* $\mathbb{V}$ *for* $p_i \in \mathcal{P}$*, every quorum* $Q_i \in \mathscr{Q}(p_i, \mathbb{V})$ *is a survivor set of* p_i*. Moreover, given* S *a survivor set of* p_i*, there exists a view* $\mathbb{V}$ *for* p_i *such that* $S \in \mathscr{Q}(p_i, \mathbb{V})$*.*

Proof. Let $Q_i \in \mathscr{Q}(p_i, \mathbb{V})$ be a quorum for p_i with $\mathbb{V}$ a view for p_i. By Definition 6.10, all processes in Q including p_i have a slice in Q. From Definition 6.5, this implies that Q is a survivor set of p_i.

Moreover, given a survivor set S of p_i, the set S consists of slices of every member of S. This means that there exists a view $\mathbb{V}$ for p_i in which S satisfies Definition 6.10 and it is a quorum for p_i. This is the view $\mathbb{V}$ defined as follows:

1. for every $p_j \in S$, $\mathbb{V}'[j] = \mathbb{F}[j]$, and
2. for every $p_j \notin S$, $\mathbb{V}'[j] = (\emptyset, \{\emptyset\})$.

□

Example 6.12. *Let us consider Example 6.3 with survivor-set systems as shown in Example 6.6. Since all the processes already know all the configurations of every other process, we have that* $\mathcal{S}_i = \mathscr{Q}(p_i, \mathbb{F})$*, with* $\mathbb{F}$ *the permissionless fail-prone system.*

Combining the quorum sets of all processes, we now obtain a *permissionless Byzantine quorum system* for $\mathbb{F}$.

Definition 6.13 (Permissionless Byzantine quorum system)**.** *A* permissionless Byzantine quorum system *for $\mathcal{P}$ and $\mathbb{F}$ is an array of collections of sets $\mathbb{Q}_{perm} = [\mathcal{Q}(p_1, \mathbb{F}), \mathcal{Q}(p_2, \mathbb{F}), \ldots]$, where $\mathcal{Q}(p_i, \mathbb{F})$ is called the* quorum system *for p_i and is determined by the quorum function $\mathcal{Q}$.*

Observe that our notion of a quorum system differs from that in the existing literature [9], [26], [27]. In particular, standard Byzantine quorum systems are defined through a pair-wise intersection among quorums. This is possible in scenarios where the full system membership is known to every process. However, in permissionless settings, this requirement cannot as clearly be achieved globally.

Definition 6.14 (Current quorum system)**.** *Let $\mathbb{V}$ be the view representing the assumptions that a process p_i has learned so far. Then the* current quorum system of p_i *is the set $\mathcal{Q}(p_i, \mathbb{V})$. Moreover, a set of processes Q is a* current quorum *of p_i if and only if $Q \in \mathcal{Q}(p_i, \mathbb{V})$; we also say that p_i* has a quorum *Q.*

Note that, in this model, each process has its own set of quorums and the set of quorums of a process changes throughout an execution as the process learns the assumptions of more processes. Importantly, note that faulty processes may lie about their configuration and influence the quorums of correct processes. In an execution e with faulty set A, a correct process p_i might have a view in which the assumptions of processes in A are arbitrary because processes in A lied about their assumptions. However, processes outside A do not lie about their assumptions. We capture this with the following definition.

Definition 6.15 (T-resilient view)**.** *Given a set of processes T, we say that a view $\mathbb{V}$ is* T-resilient *if and only if for every process $p_i \notin T$, either $\mathbb{V}[i] = \perp$ or $\mathbb{V}[i] = \mathbb{F}[i]$.*

Intuitively, a correct process p_i will either not have heard from $p_j \notin A$ or it will have the correct assumption for p_j. Thus, p_i's view is A-resilient at all times in execution e.

As we said, processes in A may lie about their assumptions causing quorums to contain unreliable slices. Moreover, processes in A may aim at preventing intersection among quorums of correct processes. In the following definition we characterize the notion of worst-case view, i.e.,

when faulty processes gossip only empty configurations. By doing so, quorums of correct processes will contain fewer members, increasing the chances of an empty intersection among them.

Definition 6.16 (Worst-case view)**.** *Given a set of processes T, the* worst-case view *with respect to T is the view $\mathbb{V}^*_T$ such that:*

1. *for every $p_i \in \mathcal{P} \setminus T$, $\mathbb{V}^*_T[i] = \mathbb{F}[i]$, and*
2. *for every $p_i \in T$, $\mathbb{V}^*_T[i] = (\emptyset, \{\emptyset\})$.*

Finally, every quorum for a process $p_i \notin A$ in a A-resilient view contains a quorum for p_i in a worst-case view with respect to A. This is shown in the following lemma.

Lemma 6.17. *Consider a set of processes T, a T-resilient view $\mathbb{V}$, and a process $p_i \notin T$. Moreover, let us assume that processes in T may lie about their assumptions. For every quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$, there exists a quorum $Q'_i \in \mathcal{Q}(p_i, \mathbb{V}^*_T)$ such that $Q'_i \subseteq Q_i$.*

Proof. Let T be a set of processes, $\mathbb{V}$ be a T-resilient view, p_i a process not in T. Since $\mathbb{V}$ is a T-resilient view, for every $p_j \notin T$ it holds either $\mathbb{V}[j] = \bot$ or $\mathbb{V}[j] = \mathbb{F}[j]$. However, processes in T may lie about their assumptions and, because of that, the view of process $p_i \notin T$ may contain arbitrary configurations received from processes in T.

If $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ is a quorum for p_i in $\mathbb{V}$, then Q_i might contain slices of processes in T which are derived from false assumptions. One can easily show that by starting from a T-compatible view and by removing the configurations received by processes in T, it is possible to obtain the corresponding worst-case view. By removing configurations from $\mathbb{V}$, also Q_i becomes smaller, i.e., with less members, obtaining a quorum $Q'_i \subseteq Q_i$. In fact, by removing from Q_i a slice $\overline{F}_j$ of a process $p_j \in T$, also slices of other processes in $\overline{F}_j$ might get removed in order for Definition 6.10 to be satisfied on Q'_i. This proves the lemma. □

6.4 Leagues

We now define the notion of a *league*. In Section 6.7 we show how a league allows to implement Bracha broadcast.

Definition 6.18 (League)**.** *A set of processes L is a* league *for the quorum function $\mathcal{Q}$ if and only if the following properties hold:*

Consistency: *For every set $T \subseteq \mathcal{P}$ tolerated by L, for every two T-resilient views $\mathbb{V}$ and $\mathbb{V}'$, for every two processes $p_i, p_j \in L \setminus T$, and for every two quorums $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ and $Q_j \in \mathcal{Q}(p_j, \mathbb{V}')$ it holds $(Q_i \cap Q_j) \setminus T \neq \emptyset$.*

Availability: *For every set $T \subseteq \mathcal{P}$ tolerated by L and for every $p_i \in L \setminus T$, there exists a quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{F})$ for p_i such that $Q_i \subseteq L \setminus T$.*

If we consider an execution e tolerated by a league L, where A is the set of faulty processes, the consistency property of L implies that, at any time, any two quorums of correct processes in L have some correct process in common. This is similar to the consistency property of symmetric and asymmetric Byzantine quorum systems [9], [26].

Moreover, since the set of faulty processes A is tolerated by L, by the availability property of L, every correct process in L has a quorum in $\mathbb{F}$ consisting of only correct processes.

Example 6.19. *Observe that the set $\mathcal{P}$ as introduced in Example 6.3 is a league. In fact, for every set T tolerated by $\mathcal{P}$, i.e., $\emptyset$, $\{p_1\}, \{p_4\}$ and $\{p_1, p_4\}$, for every two processes $p_i, p_j \in \mathcal{P} \setminus T$ and for every two quorums $Q_i \in \mathcal{S}_i$ and $Q_j \in \mathcal{S}_j$ as in Example 6.12, it holds $(Q_i \cap Q_j) \setminus T \neq \emptyset$, and for every $p_i \in \mathcal{P} \setminus T$, there exists a quorum $Q_i \in \mathcal{S}_i$ such that $Q_i \subseteq \mathcal{P} \setminus T$.*

The following lemma shows that the union of two intersecting leagues L_1 and L_2 is again a league, assuming that for every set T tolerated by both the leagues, L_1 and L_2 have a common process not in T.

Lemma 6.20. *If L_1 and L_2 are two leagues such that $L_1 \cap L_2 \neq \emptyset$ and such that for every set T tolerated by $L_1 \cup L_2$, there exists a process $p_k \in (L_1 \cap L_2) \setminus T$, then $L_1 \cup L_2$ is a league.*

Proof. Let L_1 and L_2 be two leagues such that $L_1 \cap L_2 \neq \emptyset$. For every T tolerated by $L_1 \cup L_2$ (and so, tolerated by L_1 and L_2, independently), for every $p_i \in L_1 \setminus T$ and $p_j \in L_2 \setminus T$, for every two T-resilient views $\mathbb{V}$ and $\mathbb{V}'$ for p_i and p_j, respectively, let $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ and $Q_j \in \mathcal{Q}(p_j, \mathbb{V}')$ be two quorums for p_i and p_j, respectively. Observe that, by assumption, for every tolerated set T by $L_1 \cup L_2$ there exists a process $p_k \in (L_1 \cap L_2) \setminus T$. Let $p_k \in L_1 \cap L_2$ and let $Q_k \in \mathcal{Q}(p_k, \mathbb{V})$ be a quorum for p_k such that $Q_k \subseteq L_1$, according to availability property of L_1. From consistency property of L_2, $(Q_k \cap Q_j) \setminus T \neq \emptyset$ and every process in this intersection belongs to L_1. Observe that, Q_j is a quorum for every of its member. This implies that Q_j is a quorum for every process in $(Q_k \cap Q_j) \setminus T$ and

every process in $(Q_k \cap Q_j) \setminus T$ has quorum in L_1. Moreover, $(Q_k \cap Q_i) \setminus T \neq \emptyset$. It follows that $(Q_i \cap Q_j) \setminus T \neq \emptyset$.

Finally, by availability property of L_1 and L_2, for every tolerated set T by L_1 and L_2 and for every process $p_i \in L_1 \setminus T$ and $p_j \in L_2 \setminus T$, eventually there exists a quorum $Q_i \in (p_i, \mathbb{F})$ for p_i and a quorum $Q_j \in \mathcal{Q}(p_j, \mathbb{F})$ for p_j such that $Q_i \subseteq L_1 \setminus T$ and $Q_j \subseteq L_2 \setminus T$, respectively. If $p_i = p_j \in L_1 \cap L_2$, then there exists a quorum Q_i for p_i such that $Q_i \subseteq (L_1 \cup L_2) \setminus T$. □

In the following lemma we show that we can characterize the consistency property of a league just by considering worst-case views. Intuitively, this result relies on the observation that every T-resilient view can be seen as extensions of worst-case views with respect to $T \subseteq \mathcal{P}$, in the sense that a T-resilient view can be obtained by starting from a worst-case view with respect to T and by considering the non-empty configurations received by processes in T.

Lemma 6.21. *The consistency property a league L holds if and only if for every set $T \subseteq \mathcal{P}$ tolerated by L, for every two worst-case views $\mathbb{V}^*_T$ and $\mathbb{V}'^*_T$ with respect to T, for every two processes $p_i, p_j \in L \setminus T$, and for every two quorums $Q_i \in \mathcal{Q}(p_i, \mathbb{V}^*_T)$ and $Q_j \in \mathcal{Q}(p_j, \mathbb{V}'^*_T)$ it holds $(Q_i \cap Q_j) \setminus T \neq \emptyset$.*

Proof. Let us assume that the consistency property of a league L holds. Since the property holds for every pair of views, it must hold also for worst-case views. The implication easily follows.

Let us now assume that for every set $T \subseteq \mathcal{P}$ tolerated by L, for every two worst-case views $\mathbb{V}^*_T$ and $\mathbb{V}'^*_T$ with respect to T, for every two processes $p_i, p_j \in L \setminus T$, and for every two quorums $Q_i \in \mathcal{Q}(p_i, \mathbb{V}^*_T)$ and $Q_j \in \mathcal{Q}(p_j, \mathbb{V}'^*_T)$ it holds $(Q_i \cap Q_j) \setminus T \neq \emptyset$. Observe that, given a quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{V}^*_T)$ for p_i in a worst-case view $\mathbb{V}^*_T$, all the quorums obtained by also considering all the possible configurations received from processes in T that are not in $\mathbb{V}^*_T$ do contain Q_i. Moreover, there cannot exist a T-resilient view that does not consist of configurations of a worst-case view with respect to T. If this was the case, then by removing configurations received from processes in T one would obtain a worst-case view with respect to T, reaching a contradiction. So, all the quorums obtained from T-resilient views will also intersect in processes that are not contained in T. □

Now we show how a league can be abstracted and defined without considering views. This will be useful in Section 6.5 when we compare

our model with other permissionless models. First, we introduce the following definitions.

Definition 6.22 (Inclusive up to)**.** *A set $I \subseteq \mathcal{P}$ is* inclusive up to *a set $T \subseteq \mathcal{P}$ if and only if for every $p_i \in I \setminus T$, process p_i has a slice in I.*

If we consider an execution e with set of faulty processes A then a set of processes I is inclusive up to A if and only if every correct process in I has a slice contained in I.

Definition 6.23 (Rooted at)**.** *A set $R \subseteq \mathcal{P}$ is* rooted at a process p_i *if and only if p_i has a slice in R. A set $R \subseteq \mathcal{P}$ is* rooted in a set $T' \subseteq \mathcal{P}$ *whenever R is rooted at a member of T'.*

Lemma 6.24. *If $\mathbb{V}$ is a T-resilient view and $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ for some process p_i, then Q_i is inclusive up to T and rooted at p_i.*

Proof. If $\mathbb{V}$ is a T-resilient view then, by Definition 6.15, processes outside T do not lie about their assumptions. By definition of a quorum Q_i for a process p_i in a view $\mathbb{V}$, every process in Q_i, and so in $Q_i \setminus T$, has a slice in Q_i. This implies that Q_i is inclusive up to T and rooted at p_i. □

In the following lemma we show that given a set of processes T tolerated by $L \subseteq \mathcal{P}$, for every set of processes I inclusive up to T and rooted at $p_i \in L \setminus T$ it is possible to find a T-resilient view in which I is a quorum for p_i. This view is a worst-case view with respect to T.

Lemma 6.25. *Let L be a set of processes. For every set $T \subseteq \mathcal{P}$ tolerated by L, if $I \subseteq \mathcal{P}$ is a set inclusive up to T and rooted at $p_i \in L \setminus T$, then there is a T-resilient view in which I is a quorum for p_i.*

Proof. Let $T \subseteq \mathcal{P}$ be a tolerated set by a set of processes L and let $I \subseteq \mathcal{P}$ be a set inclusive up to T and rooted at $p_i \in L \setminus T$. This implies that p_i and every other process $p_j \in I \setminus T$ have a slice in I. Let us consider the worst-case view $\mathbb{V}^*_T$ with respect to T. Clearly, $\mathbb{V}^*_T$ is T-resilient. This implies that, in $\mathbb{V}^*_T$, the set I is a quorum for p_i, i.e., $I \in \mathcal{Q}(p_i, \mathbb{V}^*_T)$. □

Remark 6.26. *Observe that given a set of processes L and a worst-case view $\mathbb{V}^*_T$ with respect to a set $T \subseteq \mathcal{P}$ tolerated by L, every quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{V}^*_T)$ for $p_i \in L \setminus T$ is inclusive up to T and rooted at p_i.*

Moreover, given the set $\mathcal{I}$ of all the sets $I \subseteq \mathcal{P}$ inclusive up to T and rooted at $p_i \in L \setminus T$, the set $\mathcal{I}$ contains all the quorums $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$

for every T-resilient view $\mathbb{V}$. In fact, by Definition 6.22, given a set of processes I inclusive up to a set of processes T, the requirement of having a slice in I is only for processes in $I \setminus T$, leaving processes in $T \cap I$ with no requirements on the choice of their slices.

However, given a T-resilient view $\mathbb{V}$, by Definition 6.10, a quorum Q_i for p_i requires instead every process in Q_i to have a slice contained in Q_i. This means that given a T-resilient view $\mathbb{V}$, quorum Q_i for p_i is contained in $\mathcal{I}$, being a special case of inclusive set up to T.

Lemma 6.27. *The consistency property of a league L holds if and only if for every set $T \subseteq \mathcal{P}$ tolerated by L, and for every two sets $I \subseteq \mathcal{P}$ and $I' \subseteq \mathcal{P}$ that are rooted at $L \setminus T$ and inclusive up to T it holds $(I \cap I') \setminus T \neq \emptyset$.*

Proof. Let us assume that the consistency property of a league L holds. Suppose by contradiction that there is a set $T \subseteq \mathcal{P}$ tolerated by L and two sets $I \subseteq \mathcal{P}$ and $I' \subseteq \mathcal{P}$ that are inclusive up to T and rooted at $L \setminus T$ in p_i and p_j, respectively, such that $(I \cap I') \setminus T = \emptyset$.

By Lemma 6.25, there are a T-resilient view $\mathbb{V}$ in which I is a quorum for p_i and a T-resilient view $\mathbb{V}'$ in which I' is a quorum for p_j and we reached a contradiction.

Let us now assume that for every set $T \subseteq \mathcal{P}$ tolerated by L, and for every two sets $I \subseteq \mathcal{P}$ and $I' \subseteq \mathcal{P}$ that are inclusive up to T and rooted at $L \setminus T$ it holds $(I \cap I') \setminus T \neq \emptyset$.

Let $\mathcal{I}$ and $\mathcal{I}'$ be the sets of all the sets $I \subseteq \mathcal{P}$ and $I' \subseteq \mathcal{P}$ inclusive up to T and rooted at $p_i \in L \setminus T$ and $p_j \in L \setminus T$, respectively. The proof follows from the reasoning in Remark 6.26: for every two T-resilient views $\mathbb{V}$ and $\mathbb{V}'$, every quorum $Q_i \in \mathscr{Q}(p_i, \mathbb{V})$ for p_i is contained in $\mathcal{I}$ and every quorum $Q_j \in \mathscr{Q}(p_j, \mathbb{V})$ for p_j is contained in $\mathcal{I}'$. □

Lemma 6.28. *The availability property of a league L holds if and only if for every set $T \subseteq \mathcal{P}$ tolerated by L, every member of $L \setminus T$ has a survivor set in $L \setminus T$.*

Proof. Let us assume that the availability property of a league L holds, i.e., for every set of processes T tolerated by L and for every $p_i \in L \setminus T$, there exists a quorum $Q_i \in \mathscr{Q}(p_i, \mathbb{F})$ for p_i such that $Q_i \subseteq L \setminus T$. This means that, by Definition 6.5, every process in $L \setminus S$ has a survivor set in $L \setminus S$.

Let us now assume that for every set $T \subseteq \mathcal{P}$ tolerated by L, every member of $L \setminus T$ has a survivor set S in $L \setminus T$. Let p_i be a process

in L, by Definition 6.10 we have that $S \in \mathcal{Q}(p_i, \mathbb{F})$ for p_i. The proof follows. □

6.5 Comparison with other models

In this section we compare our model with the symmetric model based on fail-prone systems [26], with the asymmetric model [9], [15], with the federated Byzantine agreement system model [28], and with the personal Byzantine quorum system model [24].

6.5.1 Comparison with symmetric fail-prone systems

We show that symmetric fail-prone systems and quorums (Chapter 3) can be understood as a special case of our model, when every process knows the entire system and assumes the same, global fail-prone system.

Recall that, in a symmetric system, the failures that are tolerated by the processes are all possible subsets of fail-prone sets in $\mathcal{F}$ and we have $P_i = \mathcal{P}$ for every $p_i \in \mathcal{P}$. Every process also knows the global quorum system.

We define a bijective function f between the set of fail-prone systems and a subset of PFPS such that $f(\mathcal{F}) = [(\mathcal{P}, \mathcal{F}), \ldots, (\mathcal{P}, \mathcal{F})]$ with n repetitions and we notice that in symmetric fail-prone systems there is only one view, namely $\mathbb{V} = f(\mathcal{F})$.

We define the quorum function $\mathcal{Q} : \mathcal{P} \times \Upsilon \to 2^{\mathcal{P}}$ such that for every process $p_i \in \mathcal{P}$, $\mathcal{Q}(p_i, f(\mathcal{F})) = \overline{\mathcal{F}}$. Observe that any set in $\overline{\mathcal{F}}$ is a slice of every process $p_i \in \mathcal{P}$ according to Definition 6.4. In the next theorem we consider this quorum function and show that, given some assumptions on $\mathcal{F}$, any set in $\overline{\mathcal{F}}$ is also a quorum for every process $p_i \in \mathcal{P}$ according to Definition 6.10.

Theorem 6.29. *Let $\mathcal{P}$ be the set of all processes and $\mathcal{F}$ the fail-prone system for $\mathcal{P}$. Then $Q^3(\mathcal{F})$ holds if and only if $\mathcal{P}$ is a league for the quorum function $\mathcal{Q}$ in $f(\mathcal{F})$.*

Proof. Let us first assume that $Q^3(\mathcal{F})$ holds. This means that for every $F_1, F_2, F_3 \in \mathcal{F}$, $\mathcal{P} \not\subseteq F_1 \cup F_2 \cup F_3$. It follows that $\overline{\mathcal{F}}$ is a quorum system for $\mathcal{F}$. Consistency property of $\overline{\mathcal{F}}$ implies that for every tolerated set F, for every two processes p_i and p_j in $\mathcal{P} \setminus F$ and for every two quorums

$Q_i \in \mathcal{Q}(p_i, f(\mathcal{F}))$ and $Q_j \in \mathcal{Q}(p_j, f(\mathcal{F}))$ for p_i and p_j, respectively, it holds that $(Q_i \cap Q_j) \setminus T \neq \emptyset$.

The availability property of $\overline{\mathcal{F}}$ implies that for every set $F \in \mathcal{F}$ tolerated by $\mathcal{P}$, every process $p_i \in \mathcal{P} \setminus F$ has a quorum in $\mathcal{P} \setminus F$: given F, there exists a quorum $Q \in \mathcal{Q}(p_i, f(\mathcal{F}))$ such that $Q \cap F = \emptyset$ and $Q \subseteq \mathcal{P} \setminus F$. It follows that $\mathcal{P}$ is a league for the quorum function $\mathcal{Q}$ in $f(\mathcal{F})$.

Let us now assume that $\mathcal{P}$ is a league for the quorum function $\mathcal{Q}$ in $f(\mathcal{F})$. The consistency property of $\mathcal{P}$ implies that for every T tolerated by $\mathcal{P}$ (which are all the sets in $\mathcal{F}$), for every two processes p_i and p_j in $\mathcal{P} \setminus T$, for every two quorums $Q_i \in \overline{\mathcal{F}}$ and $Q_j \in \overline{\mathcal{F}}$ for p_i and p_j, respectively, it holds $(Q_i \cap Q_j) \setminus T \neq \emptyset$. Moreover, by availability property of $\mathcal{P}$ there exists a quorum in $\mathcal{P} \setminus T$ (which is the same for every process $p_i \notin T$). This implies that, for every fail-prone set $F \in \mathcal{F}$, there is a quorum Q_i such that $Q_i \cap F = \emptyset$.

These two facts imply that $\overline{\mathcal{F}}$ is a symmetric Byzantine quorum system for $\mathcal{F}$ and so $Q^3(\mathcal{F})$ holds. □

6.5.2 Comparison with asymmetric fail-prone systems

Recall that, in the asymmetric model [15] (Chapter 4), every process is free to express its own trust assumption about the processes in one common globally known system through a subjective fail-prone system.

Let $\mathcal{P}$ be a set of processes in the asymmetric model and $\mathbb{F}' = [\mathcal{F}'_1, \ldots, \mathcal{F}'_n]$ be an asymmetric fail-prone system. Define the function g from asymmetric fail-prone systems to PFPS such that $g(\mathbb{F}') = [(\mathcal{P}, \mathcal{F}'_1), \ldots, (\mathcal{P}, \mathcal{F}'_n)]$. Observe that, in an asymmetric system, the failures that may be tolerated by the processes are possible subsets of fail-prone sets in the fail-prone systems of $\mathbb{F}'$ and $P_i = \mathcal{P}$ for every $p_i \in \mathcal{P}$. Moreover, as in the symmetric model, there is only one view, which is $\mathbb{V} = g(\mathbb{F}')$.

We define the quorum function $\mathcal{Q} : \mathcal{P} \times \Upsilon \rightarrow 2^{\mathcal{P}}$ such that for every guild $\mathcal{G} \subseteq 2^{\mathcal{P}}$, if $p_i \in \mathcal{G}$ then $\mathcal{Q}(p_i, g(\mathbb{F}')) = \{\mathcal{G}, \mathcal{P}\}$, otherwise $\mathcal{Q}(p_i, g(\mathbb{F}')) = \{\mathcal{P}\}$.

Observe that a quorum in the asymmetric model is a slice according to Definition 6.4 and, for every process $p_i \in \mathcal{P}$, every set in $\mathcal{Q}(p_i, g(\mathbb{F}'))$ is a quorum according to Definition 6.10.

Through the following theorem we establish the relationship between the asymmetric model and the permissionless model.

Theorem 6.30. *Let us consider an asymmetric model among a set $\mathcal{P}$ of processes with asymmetric fail-prone system $\mathbb{F}'$. If $B^3(\mathbb{F}')$ holds and $\mathcal{P}$ tolerates some sets $T \subseteq \mathcal{P}$, then there exists a quorum function $\mathscr{Q}$ such that $\mathcal{P}$ is a league in $g(\mathbb{F}')$.*

Proof. Let us assume that $\mathcal{P}$ tolerates some sets $T \subseteq \mathcal{P}$ and let us consider the quorum function $\mathscr{Q}$ defined in this section in the context of the asymmetric model. This means that, for every set T tolerated by $\mathcal{P}$, every process $p_i \in \mathcal{P} \setminus T$ has a slice contained in $\mathcal{P} \setminus T$. This implies that in every execution in which T is the set of faulty processes, every process in $\mathcal{P} \setminus T$ is wise and $\mathcal{P} \setminus T$ is a guild.

Moreover, let us also assume that $B^3(\mathbb{F}')$ holds. This implies the existence of an asymmetric Byzantine quorum system $\mathbb{Q}'$ such that for every set T tolerated by $\mathcal{P}$, for every two processes p_i and p_j in $\mathcal{P} \setminus T$ and for every two quorums $Q_i \in \mathcal{Q}'_i$ and $Q_j \in \mathcal{Q}'_j$ for p_i and p_j, respectively, it holds that $(Q_i \cap Q_j) \setminus T \neq \emptyset$.

Observe that the set $\mathcal{P} \setminus T \in \mathscr{Q}(p_i, g(\mathbb{F}'))$ is a quorum in the permissionless model for every $p_i \in \mathcal{P} \setminus T$ according to Definition 6.10. This implies that $\mathcal{P}$ satisfies availability property of a league.

Finally, for the consistency property observe that for every process $p_i \in \mathcal{P}$, the set system $\mathscr{Q}(p_i, g(\mathbb{F}'))$ satisfies Definition 6.10; by construction we have at most only two quorums for every p_i which are $\mathcal{P}$ and $\mathcal{P} \setminus T$ both satisfying Definition 6.10.

Consistency of $\mathbb{Q}'$ implies intersection among the quorums in $\mathscr{Q}(p_i, g(\mathbb{F}'))$, for every process in $\mathcal{P} \setminus T$.

It follows that $\mathcal{P}$ is a league for the quorum function $\mathscr{Q}$ in $g(\mathbb{F}')$. □

Theorem 6.30 shows a relation between the asymmetric model and the permissionless model. In particular, if $B^3(\mathbb{F}')$ holds and $\mathcal{P}$ tolerates some sets T, then the quorum function $\mathscr{Q}$ makes $\mathcal{P}$ a league.

However, we could have scenarios in which only a subset of $\mathcal{P}$ tolerates some sets T. In particular, we have the following result.

Lemma 6.31. *Let $\mathcal{P} = \{p_1, \ldots, p_n\}$ be a set of processes, $\mathbb{F}'$ be an asymmetric fail-prone system over $\mathcal{P}$ and $g(\mathbb{F}')$ the corresponding PFPS as described in the text. Moreover, let us consider an execution e with set of faulty processes A with guild $\mathcal{G}$. Then, $\mathcal{G}$ is the only set that tolerates e.*

Proof. By definition of guild, every process in $\mathcal{G}$ is wise and has a quorum contained in $\mathcal{G}$. Observe that, given a wise process p_i, there exists a fail-prone set $F \in \mathcal{F}'_i$ in $\mathbb{F}'$ such that $A \subseteq F$. Moreover, a quorum Q_i for

p_i in the asymmetric model satisfies Definition 6.4 and it is then a slice of p_i. This implies that every process in $\mathcal{G}$ has its assumptions satisfied according to Definition 6.2. Moreover, every process in $\mathcal{G}$ has a slice contained in $\mathcal{G}$. □

In the following lemma we characterize a link between the notion of a guild, in a given execution, and a league.

Lemma 6.32. *Let us consider an asymmetric Byzantine quorum system $\mathbb{Q}'$ and a guild $\mathcal{G}$ in any execution with set of faulty processes A. Then, $\mathcal{G}$ is a league for the quorum function $\mathscr{Q}$ in $g(\mathbb{F}')$.*

Proof. The result follows from Theorem 6.30 by applying the same reasoning with $\mathcal{G}$ instead of $\mathcal{P} \setminus T$ as a guild. □

In the following lemma we show a scenario where no asymmetric Byzantine quorum systems exist but it is possible to find a league for $\mathscr{Q}$ in $g(\mathbb{F}')$.

Lemma 6.33. *There exists an asymmetric fail-prone system $\mathbb{F}'$ such that:*

1. *there is no asymmetric Byzantine quorum system for $\mathbb{F}'$, but*
2. *there exists a quorum function $\mathscr{Q}$ that make $\mathcal{P}$ a league in $g(\mathbb{F}')$.*

Proof. We prove this lemma through an example with four processes. Consider an asymmetric fail-prone system $\mathbb{F}'_4$ over four processes p_1, p_2, p_3, and p_4 with $\mathcal{F}'_1 = \{\{p_3, p_4\}\}$, $\mathcal{F}'_2 = \{\{p_1, p_4\}\}$, $\mathcal{F}'_3 = \{\{p_1, p_4\}\}$, and $\mathcal{F}'_4 = \{\{p_1, p_2\}\}$, as in Example 6.3.

Observe that, by the availability property of an asymmetric Byzantine quorum system, p_1 must have a quorum in $\{p_1, p_2\}$ and p_4 must have a quorum in $\{p_3, p_4\}$. Since $\{p_1, p_2\}$ and $\{p_3, p_4\}$ are disjoint, it is impossible to satisfy the consistency property. Thus, there does not exist any asymmetric Byzantine quorum system for $\mathbb{F}'_4$. Another way to see this is by observing that $B^3(\mathbb{F}'_4)$ does not hold: $\{p_3, p_4\} \cup \{p_1, p_2\} = \mathcal{P}$.

However, as shown in Example 6.19, $\mathcal{P}$ is a league in $g(\mathbb{F}'_4)$. So, there is no asymmetric Byzantine quorum system for $\mathbb{F}'_4$ but $\mathscr{Q}$ makes $\mathcal{P}$ a league in $g(\mathbb{F}'_4)$. □

6.5.3 Comparison with federated Byzantine agreement systems

The federated Byantine agreement system (FBAS) model has been introduced by Mazières [28] in the context of the Stellar white paper. Differently from the models presented before in this section, the FBAS model is a permissionless model, where processes, each with an initial set of known processes, continuously discover new processes. In a FBAS, every process p_i chooses a set of slices, which are sets of processes sufficient to convince p_i of agreement and a set of processes Q_i is a quorum for p_i whenever p_i has at least one slice inside Q_i and every member of Q_i has a slice that is a subset of Q_i. In particular, a quorum Q_i is a quorum for every of its members.

However, despite the permissionless nature of a FBAS, a global intersection property among quorums is required for the analysis of the Stellar Consensus Protocol (SCP), and the scenario with disjoint quorums is not considered by Mazières.

A central notion in FBAS is that of *intact* set; given a set of processes $\mathcal{P}$, an execution with set of faulty processes A and a set of correct processes $\mathcal{W} = \mathcal{P} \setminus A$, a set of processes $\mathcal{I} \subseteq \mathcal{W}$ is an *intact set* [18], [24] when the following conditions hold:

1. Consistency: for every two processes p_i and p_j in $\mathcal{I}$ and for every two quorums Q_i and Q_j for p_i and p_j, respectively, $Q_i \cap Q_j \cap \mathcal{I} \neq \emptyset$; and

2. Availability: $\mathcal{I}$ is a quorum for every of its members.

Every process in $\mathcal{I}$ is called *intact*, while every process in $\mathcal{P} \setminus \mathcal{I}$ (correct or faulty) is called *befouled* and some properties of the Stellar Consensus Protocol are guaranteed only for intact processes. Moreover, the union of two intersecting intact sets is an intact set. Finally, by requiring a system-wide intersection among quorums (as in the case of the SCP) one obtains an unique intact set (Lemma 34, [18]).

We first show that our model generalizes the FBAS model by showing that a quorum in FBAS satisfies Definition 6.10.

In FBAS a notion of fail-prone system is missing and definitions are given with respect to an execution with a fixed set of faulty processes A. However, because processes define slices, an implicit fail-prone system for every process can be derived.

In particular, given a set of processes $\mathcal{P} = \{p_1, p_2, \ldots\}$, every process in $\mathcal{P}$ defines its slices based on a known subset $P_i \subseteq \mathcal{P}$ by p_i and S_i is a slice for $p_i \in \mathcal{P}$ if and only if $p_i \in S_i$ and $S_i \subseteq P_i$ [18]. Let $\mathcal{S}_i \subseteq 2^{\mathcal{P}}$ be the set of slices of p_i, we can derive the following definition.

Definition 6.34. *(Federated fail-prone system) A set $F \subseteq \mathcal{P}$ is a* fail-prone set *of p_i if and only if there exists a slice $S_i \in \mathcal{S}_i$ of p_i such that $F = P_i \setminus S_i$. The set $\mathcal{F}_i'' \subseteq 2^{\mathcal{P}}$ of all the fail-prone sets of p_i is called* fail-prone system *of p_i. Finally, we call the set $\mathbb{F}'' = [(P_1, \mathcal{F}_1''), (P_2, \mathcal{F}_2''), \ldots]$ the* federated fail-prone system.

In a FBAS, processes discover other processes' slices during an execution and so p_i implicitly learns other processes' federated fail-prone sets. Moreover, correct processes do not lie about their slices [18], [28].

It easy to observe that given different sets of slices received from different processes, Definition 6.34 implies Definition 6.8, obtaining a notion of view $\mathbb{V}$ in the FBAS model, and, because correct processes do not lie about their slices, Definition 6.15. We define the set Υ' to be the set of all the possible views in the FBAS model.

Given the notion of view in the FBAS model, we define the quorum function $\mathscr{Q} : \mathcal{P} \times \Upsilon' \to 2^{\mathcal{P}}$ such that $\mathscr{Q}(p_i, \mathbb{V})$ contains all the sets Q_i, called quorums, with $p_i \in Q_i$ and such that every process $p_j \in Q_i$ has a slice in Q_i. So, a quorum as defined by Mazières [28] satisfies Definition 6.10. Finally, in the FBAS model we introduce the notion of survivor set as defined in Definition 6.5.

In the following theorem we show that, by assuming a stronger consistency property for a league L, i.e., that the intersection among any two quorums of any two correct processes in the league contains some correct member of the league, then L is an intact set in every execution tolerated by L.

Theorem 6.35. *Let L be a league for the quorum function $\mathscr{Q}$ and let us assume that for every set $T \subseteq \mathcal{P}$ tolerated by L, for every two T-resilient views $\mathbb{V}$ and $\mathbb{V}'$, for every two processes $p_i, p_j \in L \setminus T$, and for every two quorums $Q_i \in \mathscr{Q}(p_i, \mathbb{V})$ and $Q_j \in \mathscr{Q}(p_j, \mathbb{V}')$ it holds $(Q_i \cap Q_j \cap L) \setminus T \neq \emptyset$, then L is an intact set for every every set $T \subseteq \mathcal{P}$ tolerated by L.*

Proof. Let L be a league for the quorum function $\mathscr{Q}$ and let T be a set of processes tolerated by L. If for every two T-resilient views $\mathbb{V}$ and $\mathbb{V}'$, for every two processes $p_i, p_j \in L \setminus T$, and for every two quorums $Q_i \in \mathscr{Q}(p_i, \mathbb{V})$ and $Q_j \in \mathscr{Q}(p_j, \mathbb{V}')$ it holds $(Q_i \cap Q_j \cap L) \setminus T \neq \emptyset$, then

the consistency property of intact sets follows. The availability property of an intact set follows by observing that, in $\mathbb{F}$, the set $L \setminus T$ is a quorum for every of its members. □

Observe that without the stronger consistency property assumed in Theorem 6.35, since quorums of correct processes in L may intersect in correct processes (not necessarily in L), it may be the case that L is not an intact set.

6.5.4 Comparison with personal Byzantine quorum systems

The personal Byzantine quorum system (PBQS) model has been introduced by Losa, Gafni, and Mazières [24] in the context of Stellar consensus aiming at removing the system-wide intersection property among quorums required by Mazières [28] for the SCP.

In the PBQS model a quorum for p_i is a non-empty set of processes Q_i such that if Q_i is a quorum for p_i and $p_j \in Q_i$, then there exists a quorum Q_j for p_j such that $Q_j \subseteq Q_i$. In other terms, a quorum Q_i for some process p_i must contain a quorum for every one of its members.

Losa, Gafni, and Mazières [24] point out that a global consensus among processes may be impossible since the full system membership is not known by the processes, and define the notion of *consensus cluster* as a set of processes that can instead solve a *local* consensus, i.e., consensus among the processes in a consensus cluster can be solved. In particular, given an execution with set of faulty processes A, a set of correct processes $\mathcal{C}$ is a consensus cluster when the following conditions hold:

1. Consistency: for every two processes p_i and p_j in $\mathcal{C}$ and for every two quorums Q_i and Q_j for p_i and p_j, respectively, $Q_i \cap Q_j \not\subseteq A$; and

2. Availability: for every $p_i \in \mathcal{C}$ there exists a quorum Q_i for p_i such that $Q_i \subseteq \mathcal{C}$.

Losa, Gafni, and Mazières [24] prove that the union of two intersecting consensus clusters is a consensus cluster and that maximal consensus clusters are disjoint. The latter implies that maximal consensus clusters might diverge from each other.

In the following we show a relationship between the notions of league and consensus cluster. To do so, we first show that a quorum Q_i for p_i as defined in Definition 6.10 is also a quorum for p_i in the PBQS model.

Lemma 6.36. *Let $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$ be a quorum for a process p_i in a view $\mathbb{V}$ according to Definition 6.10. Then Q_i is a quorum for p_i in the PBQS model.*

Proof. Definition 6.10 implies that Q_i is a quorum for every of its members. This means that for every process $p_j \in Q_i$, the set Q_i is a quorum for p_j such that $Q_i \subseteq Q_i$. The result follows. □

In the following result we show that, given a league L, for every set $T \subseteq \mathcal{P}$ tolerated by L, the set $L \setminus T$ is a consensus cluster.

Theorem 6.37. *Let L be a league for the quorum function $\mathcal{Q}$. Then, for every set $T \subseteq \mathcal{P}$ tolerated by L, the set $L \setminus T$ is a consensus cluster.*

Proof. Let L be a league for the quorum function $\mathcal{Q}$ and let T be a set of processes tolerated by L. Lemma 6.36 implies that for every process p_i and for every view $\mathbb{V}$, all the quorums in $\mathcal{Q}(p_i, \mathbb{V})$ for p_i are quorums in the PBQS model. So, the consistency and availability properties of a league imply that $L \setminus T$ satisfies the consistency and availability properties of a consensus cluster, making $L \setminus T$ a consensus cluster. □

6.6 Permissionless shared memory

In this section we present a first application of permissionless Byzantine quorum systems by showing how to emulate shared memory, represented by a *register*.

A register stores values and can be accesses through two operations: *write*(v), parameterized by a value v belonging to a domain $\mathcal{V}$, and outputs a token ACK when it completes; and *read*, which takes no parameter and outputs a value $v \in \mathcal{V}$ upon completion. In this work we consider only a *single-writer* register, where only a designated process p_w may invoke *write*, and allow *multiple readers*, i.e., every process may execute a read operation. After a process has invoked an operation, the register may trigger an event that carries the reply from the operation. We say that the process *completes* the operation when this event occurs. Moreover, after a process has invoked an operation on a register, the process does not invoke any further operation on that register until the

previous operation completes and we say that a correct process accesses the registers in a *sequential* manner. An operation o *precedes* another operation o' in a sequence of events whenever o completes before o' is invoked. Two operations are *concurrent* if neither one of them precedes the other.

Definition 6.38 (Permissionless SWMR)**.** *A protocol for* permissionless single-writer multi-reader *register satisfies the following properties. For every league L and every execution tolerated by L:*

Termination: *If a correct process $p_i \in L$ invokes an operation on the register, p_i eventually completes the operation.*

Validity: *Every read operation of a correct process in L that is not concurrent with a write returns the last value written by a correct process in L; a read of a correct process in L concurrent with a write of a correct process in L may also return the value that is written concurrently.*

In Algorithm 11, the writer p_w waits until receiving ACK messages from all processes in a quorum $Q_w \in \mathcal{Q}(p_w, \mathbb{V})$. The reader p_r waits for a VALUE message with a value/timestamp pair from every process in a quorum $Q_r \in \mathcal{Q}(p_r, \mathbb{V})$.

The function *highestval*(S) takes a set of timestamp/value pairs S and returns the value of the pair with the largest timestamp.

Finally, the protocol uses digital signatures, with operations $sign_i$, invoked by process p_i, and $verify_i$. In particular, $sign_i$ takes a message $m \in \{0,1\}^*$ as input and returns a signature $\sigma \in \{0,1\}^*$, while $verify_i$ takes as input a signature σ and a message $m \in \{0,1\}^*$ and returns TRUE if and only if p_i *signed* the message m and obtained σ, or FALSE otherwise.

Theorem 6.39. *Algorithm 11 implement permissionless SWMR.*

Proof. Let us consider a league L and a tolerated execution e with set of faulty processes A. To prove the *termination* property, let us consider a writer p_w. By assumption, process p_w is correct in the league L and, by the availability property of L, eventually there exists a quorum $Q_w \in \mathcal{Q}(p_w, \mathbb{F})$ contained in $L \setminus A$. Therefore, p_w will receive sufficiently many ACK messages and the write will return. Let p_r be a reader in $L \setminus A$. As above, eventually there exists a quorum $Q_r \in \mathcal{Q}(p_r, \mathbb{F})$ contained in $L \setminus A$. Because the writer is correct and in the league, all the responses from processes in Q_r satisfy the checks and *read* returns.

Algorithm 11 Emulation of a permissionless SWMR regular register (Code for p_i)

1: **State**
2: ts_w: sequence number of write operations, stored only by writer p_w
3: id_r: identifier of read operations, used only by reader
4: ts, v, σ: current state stored by p_i: timestamp, value, signature

5: **upon invocation** *write*(v) **do** // if $p_i = p_w$
6: $ts_w \leftarrow ts_w + 1$
7: $\sigma \leftarrow sign_w(\text{WRITE}\|w\|ts_w\|v)$
8: send message $[\text{WRITE}, ts_w, v, \sigma, \mathbb{F}_i]$ through gossip
9: **wait for** receiving a gossiped message $[\text{ACK}, (P_j, \mathcal{F}_j)]$
10: from all processes in $\mathcal{Q}(p_w, \mathbb{V})$
11: $\mathbb{V}[j] \leftarrow (P_j, \mathcal{F}_j)$

12: **upon** receiving a gossiped message $[\text{WRITE}, ts', v', \sigma', (P_w, \mathcal{F}_w)]$
13: from p_w **do** // every process
14: **if** $ts' > ts$ **then**
15: $\mathbb{V}[w] \leftarrow (P_w, \mathcal{F}_w)$
16: $(ts, v, \sigma) \leftarrow (ts', v', \sigma')$
17: send message $[\text{ACK}, \mathbb{F}[i]]$ through gossip to p_w

18: **upon invocation** *read* **do** // if $p_i = p_r$
19: $id_r \leftarrow id_r + 1$
20: send message $[\text{READ}, id_r, \mathbb{F}[i]]$ through gossip
21: **wait for** receiving gossiped messages $[\text{VALUE}, r_j, ts_j, v_j, \sigma_j, (P_j, \mathcal{F}_j)]$
22: from all processes in $\mathcal{Q}(p_r, \mathbb{V}_r)$
23: **such that** $r_j = id_r$ **and** $verify_w(\sigma_j, \text{WRITE}\|w\|ts_j\|v_j)$
24: $\mathbb{V}[j] \leftarrow (P_j, \mathcal{F}_j)$
25: **return** $highestval(\{(ts_j, v_j) |\ j \in Q_r\}$

26: **upon** receiving a gossiped message $[\text{READ}, r, (P_r, \mathcal{F}_r)]$
27: from p_r **do** // every process
28: $\mathbb{V}_i[r] \leftarrow (P_r, \mathcal{F}_r)$
29: send message $[\text{VALUE}, r, ts, v, \sigma, \mathbb{F}_i]$ through gossip to p_r

For the *validity* property, observe that by assumption both the writer p_w and the reader p_r are correct processes in $L \setminus A$. If the writer p_w *writes* to a quorum $Q_w \subseteq \mathcal{Q}(p_w, \mathbb{V})$ for itself, and the reader p_r reads from a quorum $Q_r \subseteq \mathcal{Q}(p_r, \mathbb{V}')$ for itself, with $\mathbb{V}$ and $\mathbb{V}'$ two A-resilient views, by the consistency property of L it holds $(Q_w \cap Q_r) \setminus A \neq \emptyset$.

Hence, there is some correct process $p_i \in Q_w \cap Q_r$ that received the most recently written value from p_w and returns it to p_r.

Observe that, from the properties of the signature scheme, any value output by *read* has been written in some preceding or concurrent *write* operation. □

6.7 Permissionless reliable broadcast

In this section we show how the Bracha broadcast [5], protocol that implements Byzantine reliable broadcast, can be adapted to work in our model. First, we introduce the following definitions and results.

Definition 6.40 (Blocking set)**.** *A set $B \subseteq \mathcal{P}$ is said to* block *a process p_i if B intersects every slice of p_i.*

Definition 6.41 (Inductively blocked)**.** *Given a set of processes B, the set of processes inductively blocked by B, denoted by B^+, is the smallest set closed under the following rules:*

1. *$B \subseteq B^+$; and*
2. *if a process p_i is blocked by B^+, then $p_i \in B^+$.*

As a consequence of Definition 6.41, given an execution, the set B^+ can be obtained by repeatedly adding to it all the processes that are blocked by $B^+ \cup B$. Eventually no more processes will be added to B^+.

Moreover, given an execution e with set of faulty processes A, if a league L tolerates A, then processes in $L \setminus A$ cannot be inductively blocked by A. This is shown in the following lemma.

Lemma 6.42. *Let L be a league and T be a set tolerated by L. Then, no process in $L \setminus T$ is inductively blocked by T, i.e., $T^+ \cap (L \setminus T) = \emptyset$.*

Proof. Let us assume that $T^+ \cap (L \setminus T) \neq \emptyset$. This means that there exists a process $p_i \in L \setminus T$ that is blocked by T^+, i.e., T^+ intersects every slice of p_i, including the slice contained in the quorum $Q_i \subseteq L \setminus T$ for p_i.

Clearly, $(L \setminus T) \cap T = \emptyset$, and this means that there exists a set T' with $T' \subseteq T^+ \setminus T$ such that T' intersects every slice of p_i, including the slice contained in the quorum for p_i consisting only of correct processes in L. This means that we can find a process $p_j \in T'$ with $p_j \in L \setminus T$ and p_j blocked by T. Since L is a league, process p_j must have a slice

in $L \setminus T$. However, T cannot intersect every slice of p_j because $L \setminus T$ is disjoint from T. We reached a contradiction. □

Intuitively, starting from $A^+ = \emptyset$, we first consider the processes that are blocked by A. Trivially, every process in A is blocked by A, and so $A^+ = A$. Moreover, no process in $L \setminus A$ can be blocked by A. If this was the case, then there would exist a process $p_i \in L \setminus A$ such that A intersects all of its slices, including the slice contained in the quorum $Q_i \subseteq L \setminus A$, which we know to exist due to the availability property of L. So, only processes p_j not in $L \setminus A$ can be blocked by A. Let p_j be such process. This means that $A \cup \{p_j\} \subseteq A^+$. Now, we can repeat the same reasoning, by considering all the processes blocked by $A \cup \{p_j\}$. Again, no processes in $L \setminus A$ can be blocked by $A \cup \{p_j\}$. In fact, if $A \cup \{p_j\}$ blocked a process $p_k \in L \setminus A$, then every slice of p_k would contain p_j, including the slice contained in $L \setminus A$. However, this would imply that $p_j \in L \setminus A$ which would contradict the fact that p_j is a process not in $L \setminus A$.

In the following theorem we show that if a correct process p_i in a league L is blocked by a set B, then $B = B \cup \{p_i\}$ blocks another process $p_j \notin B \cup A$. Then, $B' = B \cup \{p_j\}$ blocks another process $p_k \notin B' \cup A$ and so on, until, eventually, every correct process in the league is blocked.

Theorem 6.43 (Cascade theorem)**.** *Consider the quorum function $\mathcal{Q}$, a league L, and a set $T \subseteq \mathcal{P}$ tolerated by L. Moreover, let us consider a process $p_i \in L \setminus T$, a T-resilient view $\mathbb{V}$ for p_i, a quorum $Q_i \in \mathcal{Q}(p_i, \mathbb{V})$, and a set $B \subseteq \mathcal{P}$ disjoint from T such that $Q_i \setminus T \subseteq B$. Then, either $L \setminus T \subseteq B$ or there exists a process $p_j \notin B \cup T$ that is blocked by B.*

Proof. It suffices to assume by contradiction that $L \setminus (B \cup T) \neq \emptyset$ and that, for every $p_j \notin B \cup T$, process p_j has a slice disjoint from B. This implies that $S = \overline{B \cup T}$ is a survivor set of every process $p_j \in S$; since $L \setminus (B \cup T) \neq \emptyset$, this includes also at least one process $p_j \in L \setminus (B \cup T)$.

Let us consider such a process $p_j \in L \setminus (B \cup T)$ and consider the view $\mathbb{V}'$ for p_j such that:

1. for every $p_k \notin T$, $\mathbb{V}'[k] = \mathbb{F}[k]$; and
2. for every $p_k \in T$, $\mathbb{V}'[k] = (\emptyset, \{\emptyset\})$.

Observe that $\mathbb{V}'$ is a T-resilient view for p_j. By Lemma 6.11, we have that $S \in \mathcal{Q}(p_j, \mathbb{V}')$. This implies that $S \cap Q_i \subseteq T$. But combined with the fact that $p_j \in L \setminus (B \cup T)$, this contradicts the consistency property of L. □

We will see how this theorem has a direct effect on the liveness of permissionless Byzantine reliable broadcast.

In a Byzantine reliable broadcast, the sender process may *broadcast* a value v by invoking *r-braodcast*(v). The broadcast primitive outputs a value v through an *r-deliver*(v) event. Moreover, the broadcast primitive presented in this section delivers only one value per instance. Every instance has an implicit label and a fixed, well-known sender p_s.

Definition 6.44 (Permissionless Byzantine reliable broadcast)**.** *A protocol for* permissionless Byzantine reliable broadcast *satisfies the following properties. For every league L and every execution tolerated by L:*

Validity: *If a correct process p_s* r-broadcasts *a value v, then all correct processes in L eventually* r-deliver *v.*

Integrity: *For any value v, every correct process* r-delivers *v at most once. Moreover, if the sender p_s is correct and the receiver is correct and in L, then v was previously* r-broadcast *by p_s.*

Consistency: *If a correct process in L* r-delivers *some value v and another correct process in L* r-delivers *some value v', then $v = v'$.*

Totality: *If a correct process in L* r-delivers *some value v, then all correct processes in L eventually* r-deliver *some value.*

We implement this primitive in Algorithms 12-13, which are derived from Bracha broadcast [5] but differs in some aspects.

In principle, the protocol follows the original one, but does not use one global quorum system known to all processes. Instead, the correct processes implicitly use the same quorum function $\mathcal{Q}$ (Definition 6.10), of which they initially only know their own entry in $\mathcal{Q}$. They discover the quorums of other processes during the execution.

Because of the permissionless nature of our model, we consider a best-effort gossip primitive to disseminate messages among processes instead of point-to-point messages.

A crucial element of Bracha's protocol is the "amplification" step, when a process receives $f + 1$ READY messages with some value v, with f the number of faulty processes in an execution, but has not sent a READY message yet. Then it also sends a READY message with v. This generalizes to receiving the same READY message with value v from a *blocking set* for p_i and is crucial for the *totality* property.

Finally, we introduce the ANY message as a message sent by a process p_i that is blocked by two sets carrying two different values v and v'. The reason for this new message lies in the consistency property of L: given an execution e with set of faulty processes A tolerated by L, the consistency property of L implies that any two quorums of any two correct processes in L have some correct process in common. Quorum intersection is then guaranteed only for correct processes in L and nothing is assured for correct processes outside L, which might gossip different values received by non-intersecting quorums. In particular, if a correct process p_i is blocked by a set containing a value v and later is blocked by a set containing a value $v' \neq v$, then p_i gossips an ANY message containing $*$. ANY messages are then ignored by correct processes in L. As we show in the Theorem 6.45, correct process in L cannot be blocked by sets containing different values.

Theorem 6.45. *Algorithms 12-13 implement permissionless Byzantine reliable broadcast.*

Proof. Observe that all the properties assume the existence of a league L and an execution e with set of faulty processes A tolerated by L.

Let us start with the *validity* property. Since the sender p_s is correct and from the availability property of L, every correct process p_i in L eventually receives a quorum Q_i for itself of ECHO messages containing the value v sent from p_s and updates its view $\mathbb{V}$ according to the views received from every process in Q_i.

Then, p_i gossips [READY, v, $\mathbb{F}[i]$] containing the value v and its current view $\mathbb{F}[i]$ unless *sent-ready* = TRUE. If *sent-ready* = TRUE then p_i already gossiped [READY, v, $\mathbb{F}[i]$].

Observe that there exists a unique value v such that if a correct process in L sends a READY message, this message contains v. In fact, if a process $p_i \in L \backslash A$ sends a READY message, either it does so after receiving a quorum Q_i for itself of ECHO messages containing v or after being blocked by a set of processes that received READY messages containing v.

In the first case, if a correct process p_i in L receives a quorum Q_i for itself of ECHO messages containing v and another correct process p_j in L receives a quorum Q_j for itself of ECHO messages containing v', by the consistency property of L, $v = v'$ and both send a READY message containing the same v.

In the second case, first observe that by Lemma 6.42 we know that $p_i \in L \setminus A$ cannot be inductively blocked by processes in A. Moreover, correct processes in L cannot be blocked by sets containing different

Algorithm 12 Permissionless Byzantine reliable broadcast protocol with sender p_s (Code for p_i) Part 1

State
  sent-echo $\leftarrow$ FALSE: indicates whether p_i has gossiped ECHO
  echos$[j] \leftarrow [\perp]$: received ECHO messages from other processes
  sent-ready $\leftarrow$ FALSE: indicates whether p_i has gossiped READY
  readys$[j] \leftarrow [\perp]$: received READY messages from other processes
  sent-any $\leftarrow$ FALSE: indicates weather p_i has gossiped [ANY, $*$, $\mathbb{F}[i]$]
  delivered $\leftarrow$ FALSE: indicates whether p_i has delivered a value
  $\mathbb{V}[j] \leftarrow$ if $i = j$ then $\mathbb{F}[i]$ else $\perp$: the current view of p_i

upon invocation *r-broadcast*(v) **do**
  send message [SEND, v, $\mathbb{F}[s]$] through gossip // only sender p_s

upon receiving a gossiped message [SEND, v, $(P_s, \mathcal{F}_s)$] from p_s **and**
  $\neg$*sent-echo* **do**
  sent-echo $\leftarrow$ TRUE
  $\mathbb{V}[s] \leftarrow (P_s, \mathcal{F}_s)$
  send message [ECHO, v, $\mathbb{F}[i]$] through gossip

upon receiving a gossiped message [ECHO, v, $(P_j, \mathcal{F}_j)$ from p_j **do**
  if *echos*$[j] = \perp$ **then**
    $\mathbb{V}[j] \leftarrow (P_j, \mathcal{F}_j)$
    echos$[j] \leftarrow v$

upon exists $v \neq \perp$ **such that** $\{p_j \in \mathcal{P} |$ *echos*$[j] = v\} \in \mathcal{Q}(p_i, \mathbb{V})$ **and**
  $\neg$*sent-ready* **do**
  sent-ready $\leftarrow$ TRUE
  send message [READY, v, $\mathbb{F}[i]$] through gossip

upon receiving a gossiped message [READY, v, $(P_j, \mathcal{F}_j)$] from p_j **do**
  if *readys*$[j] = \perp$ **then**
    $\mathbb{V}[j] \leftarrow (P_j, \mathcal{F}_j)$
    readys$[j] \leftarrow v$

values. If this was the case, then there would exist two correct processes p_i and p_j in L and two slices of p_i and p_j, respectively, in $L \setminus A$ containing two correct processes in L that received two different values v after ECHO. Again, by the consistency property of L, this is not possible.

Hence, every correct process p_j in L gossips [READY, v, $\mathbb{F}[j]$]. Eventually, every correct process p_i in L receives a quorum for itself containing

Algorithm 13 Permissionless Byzantine reliable broadcast protocol with sender p_s (Code for p_i) Part 2

28: **upon exists** $v \neq \perp$ **such that** $\{p_j \in \mathcal{P} |\ readys[j] = v\}$ blocks p_i **and**
29: $\neg$*sent-ready* **do**
30: *sent-ready* $\leftarrow$ TRUE
31: send message [READY, v, $\mathbb{F}[i]$] through gossip

32: **upon exists** $v' \neq \perp$ **such that** $\{p_j \in \mathcal{P} |\ readys[j] = v'\}$ blocks p_i **and**
33: $readys[i] = v$ **and** $v \neq v'$ **and** *sent-ready* **and** $\neg$*sent-any* **do**
34: *sent-any* $\leftarrow$ TRUE
35: send message [ANY, $*$, $\mathbb{F}[i]$] through gossip

36: **upon** receiving a gossiped message [ANY, $*$, $(P_j, \mathcal{F}_j)$] from p_j **do**
37: $\mathbb{V}[j] \leftarrow (P_j, \mathcal{F}_j)$
38: $readys[j] \leftarrow *$

39: **upon exists** $v \neq \perp$ **such that** $\{p_j \in \mathcal{P} |\ readys[j] = v\} \in \mathscr{Q}(p_i, \mathbb{V})$ **and**
40: $\neg$*delivered* **do**
41: *delivered* $\leftarrow$ TRUE
42: **output** *r-deliver(v)*

[READY, v, $(P_j, \mathcal{F}_j)$] messages and *r-delivers* v.

The first part of the *integrity* property is ensured by the *delivered* flag. For the second part observe that, by assumption, the receiver p_i is correct and in L. This implies that the quorum for p_i used to reach a decision contains some correct processes that have gossiped ECHO containing a value v they received from p_s.

For the *totality* property, let us assume that a correct process $p_i \in L$ *r-delivered* some value v. If $p_i \in L \setminus A$ *r-delivered* some value v, then it has received READY messages containing v from a quorum Q_i for itself. From Theorem 6.43 we know that exists a set B such that $Q_i \setminus A \subseteq B$ and either $L \setminus A \subseteq B$ or B blocks at least a process $p_j \in L \setminus (B \cup A)$ in an A-resilient view $\mathbb{V}'$ for p_j.

In the latter case, p_j gossips a READY message containing v and B becomes $B \cup \{p_j\}$. Observe that, by assumption, if a correct process receives a gossiped message, then eventually every other correct process receives it too. Eventually, $L \setminus A$ is covered by B and this means that every correct process in L is blocked with the same value v.

Moreover, observe that given two correct processes not in L, they may become ready for different values received from non-intersecting

quorums of ECHO messages. Because of this, if a correct process $p_j \notin L$ observes a blocking set B containing a value v' different from a value v that has previously gossiped in a READY message and such that *sent-any* = FALSE, process p_j gossips an ANY message containing the value $*$. Eventually every correct process p_i in L receives a quorum Q_i for itself of [READY, v, $(P_j, \mathcal{F}_j)$] messages and it *r-delivers* v.

Finally, for the *consistency* property notice that by the consistency property of L, every two quorums Q_i and Q_j of any two correct processes p_i and p_j in L intersect in some correct process p_k. Process p_k could then be outside L. If $p_k \notin L$ then, as seen for the totality property, it can be blocked by sets containing different values. If this is the case then p_k gossips an ANY message. Correct processes in L then ignore the values received from p_k and wait until receiving a quorum unanimously containing the same value v. Observe that, because L tolerates A, by availability property of L every correct process in L eventually receives a quorum made by correct processes in L. The consistency property then follows. □

Chapter 7

Conclusion

In this thesis we explored the notion of asymmetric trust in secure distributed systems prone to Byzantine failures. Our results offer a new approach to establish trust in systems where a global trust framework among participants cannot be imposed, such as blockchain systems.

We analyzed ways to work with systems with asymmetric trust. In particular, we showed how asymmetric trust assumptions of (possibly disjoint) systems can be composed deterministically, so that groups of participants may join each other and collaborate under a composed trust assumption with appealing properties.

We devised the first asymmetric asynchronous Byzantine consensus protocol and showed that consensus protocols with asymmetric trust can be obtained by starting from existing, well-known protocols with symmetric trust.

Finally, we extended the asymmetric trust model to cope with permissionless settings, which also resulted in a characterization with standard formalism of the model used by the Stellar blockchain. In particular, we introduced a new way of specifying trust assumptions among participants in a permissionless setting: participants not only make assumptions about failures, but also make assumptions about the assumptions of other participants. This led to formally define the notions of permissionless fail-prone system and permissionless quorum system, and to design protocols to solve known synchronization problems such as Byzantine reliable broadcast.

Although our results have opened up new ideas for approaching trust in blockchain sysyems, there are still open questions that require fur-

ther research. For instance, the role of asymmetric threshold cryptography, or how to best cope with leader-based consensus protocols in an asymmetric-trust setting. Nonetheless, our results have carried on a new way of thinking about subjective trust in secure distributed systems such as blockchain systems, and we believe that they will provide a solid foundation for future research in this area.

Bibliography

[1] I. Abraham, N. Ben-David, and S. Yandamuri, "Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement," in *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, A. Milani and P. Woelfel, Eds., ACM, 2022, pp. 381–391.

[2] I. Abraham, N. Ben-David, and S. Yandamuri, *Efficient and adaptively secure asynchronous binary agreement via binding crusader agreement*, Cryptology ePrint Archive, Paper 2022/711, 2022.

[3] O. Alpos, C. Cachin, and L. Zanolini, "How to trust strangers: Composition of byzantine quorum systems," in *40th International Symposium on Reliable Distributed Systems, SRDS 2021, Chicago, IL, USA, September 20-23, 2021*, IEEE, 2021, pp. 120–131.

[4] J. C. Benaloh and J. Leichter, "Generalized secret sharing and monotone functions," in *Proc. CRYPTO*, ser. Lecture Notes in Computer Science, vol. 403, 1988, pp. 27–35.

[5] G. Bracha, "Asynchronous byzantine agreement protocols," *Inf. Comput.*, vol. 75, no. 2, pp. 130–143, 1987.

[6] C. Cachin, R. Guerraoui, and L. E. T. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)* Springer, 2011.

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

[8] C. Cachin, G. Losa, and L. Zanolini, "Quorum systems in permissionless networks," in *26th International Conference on Principles of Distributed Systems, OPODIS 2022, December 13-15, 2022, Brussels, Belgium*, E. Hillel, R. Palmieri, and E. Rivière, Eds., ser. LIPIcs, vol. 253, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022, 17:1–17:22.

[9] C. Cachin and B. Tackmann, "Asymmetric distributed trust," in *Proc. OPODIS*, ser. LIPIcs, vol. 153, 2019, 7:1–7:16.

[10] C. Cachin and L. Zanolini, "Asymmetric asynchronous byzantine consensus," in *Data Privacy Management, Cryptocurrencies and Blockchain Technology - ESORICS 2021 International Workshops, DPM 2021 and CBT 2021, Darmstadt, Germany, October 8, 2021, Revised Selected Papers*, J. Garcìa-Alfaro, J. L. Muñoz-Tapia, G. Navarro-Arribas, and M. Soriano, Eds., ser. Lecture Notes in Computer Science, vol. 13140, Springer, 2021, pp. 192–207.

[11] C. Cachin and L. Zanolini, "Brief announcement: Revisiting signature-free asynchronous byzantine consensus," in *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, S. Gilbert, Ed., ser. LIPIcs, vol. 209, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 51:1–51:4.

[12] R. Canetti and T. Rabin, "Fast asynchronous byzantine agreement with optimal resilience," in *Proc. STOC*, 1993, pp. 42–51.

[13] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proc. USENIX*, 1999, pp. 173–186.

[14] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, 2002.

[15] I. Damgård, Y. Desmedt, M. Fitzi, and J. B. Nielsen, "Secure protocols with asymmetric trust," in *Proc. ASIACRYPT*, ser. Lecture Notes in Computer Science, vol. 4833, 2007, pp. 357–375.

[16] C. Dwork, N. A. Lynch, and L. J. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[17] H. Garcia-Molina and D. Barbará, "How to assign votes in a distributed system," *J. ACM*, vol. 32, no. 4, pp. 841–860, 1985.

[18] Á. Garcìa-Pèrez and A. Gotsman, "Federated byzantine quorum systems," in *Proc. OPODIS*, ser. LIPIcs, vol. 125, 2018, 17:1–17:16.

[19] V. Hadzilacos and S. Toueg, "Fault-tolerant broadcasts and related problems," in *Distributed Systems (2nd Ed.)* S. J. Mullender, Ed., ACM Press, 1993, pp. 97–145.

[20] M. Hirt and U. M. Maurer, "Player simulation and general adversary structures in perfect multiparty computation," *J. Cryptol.*, vol. 13, no. 1, pp. 31–60, 2000.

[21] F. P. Junqueira and K. Marzullo, "Synchronous consensus for dependent process failure," in *Proc. ICDCS*, 2003, pp. 274–283.

[22] F. P. Junqueira, K. Marzullo, M. Herlihy, and L. D. Penso, "Threshold protocols in survivor set systems," *Distributed Comput.*, vol. 23, no. 2, pp. 135–149, 2010.

[23] M. Lokhava, G. Losa, D. Mazières, *et al.*, "Fast and secure global payments with stellar," in *Proc. SOSP*, 2019, pp. 80–96.

[24] G. Losa, E. Gafni, and D. Mazières, "Stellar consensus by instantiation," in *Proc. DISC*, ser. LIPIcs, vol. 146, 2019, 27:1–27:15.

[25] D. Malkhi, K. Nayak, and L. Ren, "Flexible byzantine fault tolerance," in *Proc. ACM CCS*, 2019, pp. 1041–1053.

[26] D. Malkhi and M. K. Reiter, "Byzantine quorum systems," *Distributed Comput.*, vol. 11, no. 4, pp. 203–213, 1998.

[27] D. Malkhi, M. K. Reiter, and A. Wool, "The load and availability of byzantine quorum systems," *SIAM J. Comput.*, vol. 29, no. 6, pp. 1889–1906, 2000.

[28] D. Mazières, *The Stellar consensus protocol: A federated model for Internet-level consensus*, Stellar, available online, 2016.

[29] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous binary byzantine consensus with t ¡ n/3, o(n2) messages, and O(1) expected time," *J. ACM*, vol. 62, no. 4, 31:1–31:21, 2015.

[30] A. Mostéfaoui, H. Moumen, and M. Raynal, "Signature-free asynchronous byzantine consensus with t 2¡n/3 and o(n^2) messages," in *Proc. PODC*, 2014, pp. 2–9.

[31] M. Naor and A. Wool, "The load, capacity, and availability of quorum systems," *SIAM J. Comput.*, vol. 27, no. 2, pp. 423–447, 1998.

[32] A. Patra, A. Choudhury, and C. P. Rangan, "Asynchronous byzantine agreement with optimal resilience," *Distributed Comput.*, vol. 27, no. 2, pp. 111–146, 2014.

[33] M. C. Pease, R. E. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *J. ACM*, vol. 27, no. 2, pp. 228–234, 1980.

[34] M. O. Rabin, "Randomized byzantine generals," in *Proc. FOCS*, 1983, pp. 403–409.

[35] T. K. Srikanth and S. Toueg, "Simulating authenticated broadcasts to derive simple fault-tolerant algorithms," *Distributed Comput.*, vol. 2, no. 2, pp. 80–94, 1987.

[36] P. Tholoniat and V. Gramoli, "Formal verification of blockchain byzantine fault tolerance," in *6th Workshop on Formal Reasoning in Distributed Algorithms (FRIDA'19)*, 2019.

[37] M. Vukolic, "The origin of quorum systems," *Bulletin of the EATCS*, vol. 101, pp. 125–147, 2010.

[38] M. Yin, D. Malkhi, M. K. Reiter, G. Golan-Gueta, and I. Abraham, "Hotstuff: BFT consensus with linearity and responsiveness," in *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, PODC 2019, Toronto, ON, Canada, July 29 - August 2, 2019*, P. Robinson and F. Ellen, Eds., ACM, 2019, pp. 347–356.

Declaration of consent

on the basis of Article 18 of the PromR Phil.-nat. 19

Name/First Name: Zanolini Luca

Registration Number: 18-129-403

Study program: Computer Science

Bachelor ☐ Master ☐ Dissertation ☑

Title of the thesis: Asymmetric Trust in Distributed Systems

Supervisor: Prof. Dr. Christian Cachin

I declare herewith that this thesis is my own work and that I have not used any sources other than those stated. I have indicated the adoption of quotations as well as thoughts taken from other authors as such in the thesis. I am aware that the Senate pursuant to Article 36 paragraph 1 litera r of the University Act of September 5th, 1996 and Article 69 of the University Statute of June 7th, 2011 is authorized to revoke the doctoral degree awarded on the basis of this thesis.
For the purposes of evaluation and verification of compliance with the declaration of originality and the regulations governing plagiarism, I hereby grant the University of Bern the right to process my personal data and to perform the acts of use this requires, in particular, to reproduce the written thesis and to store it permanently in a database, and to use said database, or to make said database available, to enable comparison with theses submitted by others.

Firenze, 15/05/2023

Place/Date

Signature