Asymmetric Distributed Trust

Christian Cachin
University of Bern, Switzerland
https://crypto.unibe.ch/cc/
cachin@inf.unibe.ch

Björn Tackmann
DFINITY Foundation, Zürich, Switzerland
bjoern@dfinity.org

Abstract

Quorum systems are a key abstraction in distributed fault-tolerant computing for capturing trust assumptions. They can be found at the core of many algorithms for implementing reliable broadcasts, shared memory, consensus and other problems. This paper introduces *asymmetric Byzantine quorum systems* that model subjective trust. Every process is free to choose which combinations of other processes it trusts and which ones it considers faulty. Asymmetric quorum systems strictly generalize standard Byzantine quorum systems, which have only one global trust assumption for all processes. This work also presents protocols that implement abstractions of shared memory and broadcast primitives with processes prone to Byzantine faults and asymmetric trust. The model and protocols pave the way for realizing more elaborate algorithms with asymmetric trust.

2012 ACM Subject Classification Theory of computation → Cryptographic protocols; Software and its engineering → Distributed systems organizing principles

Keywords and phrases Quorums, consensus, distributed trust, blockchains, cryptocurrencies

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2019.7

Related Version A complete version of the paper is available at http://arxiv.org/abs/1906.09314.

Funding This work has been supported in part by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 780477 PRIViLEDGE.

Acknowledgements The authors thank Orestis Alpos, Sabine Brunner, Marko Vukolić, and Luca Zanolini for interesting discussions. Work done while both authors were at IBM Research - Zurich.

1 Introduction

Byzantine quorum systems [21] are a fundamental primitive for building resilient distributed systems from untrusted components. Given a set of nodes, a quorum system captures a trust assumption on the nodes in terms of potentially malicious protocol participants and colluding groups of nodes. Based on quorum systems, many well-known algorithms for *reliable broadcast*, *shared memory*, *consensus* and more have been implemented; these are the main abstractions to synchronize the correct nodes with each other and to achieve consistency despite the actions of the faulty, so-called *Byzantine* nodes.

Traditionally, trust in a Byzantine quorum system for a set of processes $\mathcal{P}$ has been *symmetric*. In other words, a global assumption specifies which processes may fail, such as the simple and prominent *threshold quorum* assumption, in which any subset of $\mathcal{P}$ of a given maximum size may collude and act against the protocol. The most basic threshold Byzantine quorum system, for example, allows all subsets of up to $f < n/3$ processes to fail. Some classic works also model arbitrary, non-threshold symmetric quorum systems [21, 15], but these have not actually been used in practice.

However, trust is inherently subjective. *De gustibus non est disputandum.*[1] Estimating which processes will function correctly and which ones will misbehave may depend on personal taste. A myriad of local choices influences one process' trust in others, especially because there are so many forms of "malicious" behavior. Some processes might not even be aware of all others, yet a process should not depend on unknown third parties in a distributed collaboration. How can one model asymmetric trust in distributed protocols? Can traditional Byzantine quorum systems be extended to subjective failure assumptions? How do the standard protocols generalize to this model?

In this paper, we answer these questions and introduce models and protocols for asymmetric distributed trust. We formalize *asymmetric quorum systems* for asynchronous protocols, in which every process can make its own assumptions about Byzantine faults of others. We introduce several protocols with asymmetric trust that strictly generalize the existing algorithms, which require common trust.

Our formalization takes up earlier work by Damgård et al. [10] and starts out with the notion of a fail-prone system that forms the basis of a symmetric Byzantine quorum system. A global fail-prone system for a process set $\mathcal{P}$ contains all maximal subsets of $\mathcal{P}$ that might jointly fail during an execution. In an asymmetric quorum system, every process specifies its *own* fail-prone system and a corresponding set of local quorums. These local quorum systems satisfy a *consistency condition* that ranges across all processes and a local *availability condition*, and generalize symmetric Byzantine quorum system according to Malkhi and Reiter [21].

Interest in consensus protocols based on Byzantine quorum systems has surged recently because of their application to permissioned blockchain networks [6, 1]. Typically run by a consortium, such distributed ledgers often use *Byzantine-fault tolerant (BFT)* protocols like PBFT [7] for consensus that rely on symmetric threshold quorum systems. The Bitcoin blockchain and many other cryptocurrencies, which triggered this development, started from different assumptions and use so-called permissionless protocols, in which everyone may participate. Those algorithms capture the relative influence of the participants on consensus decisions by an external factor, such as "proof-of-work" or "proof-of-stake."

A middle ground between permissionless blockchains and BFT-based ones has been introduced by the blockchain networks of Ripple (`https://ripple.com`) and Stellar (`https://stellar.org`). Their stated model for achieving network-level consensus uses subjective trust in the sense that each process declares a local list of processes that it "trusts" in the protocol.

Consensus in the *Ripple* blockchain (and for the *XRP* cryptocurrency on the *XRP Ledger*) is executed by its validator nodes. Each validator declares a *Unique Node List (UNL)*, which are nodes that a given participant trusts. Questions have been raised about the kind of decentralization offered by the Ripple protocol. *Stellar* was created as an evolution of Ripple that shares much of the same design philosophy. The Stellar consensus protocol [22] powers the *Stellar Lumen (XLM)* cryptocurrency and introduces *federated Byzantine quorum systems (FBQS)*; these bear superficial resemblance with our asymmetric quorum systems but differ technically. However, standard Byzantine quorum systems and FBQS are *not* comparable because (1) an FBQS when instantiated with the same trust assumption for all processes does not reduce to a symmetric quorum system and (2) existing protocols do not generalize to FBQS.

[1] There is no disputing about taste.

Understanding how such ideas of subjective trust, as manifested in the Ripple and Stellar blockchains, relate to traditional quorum systems is the main motivation for this work. Our protocols for asymmetric trust generalize the well-known, classic algorithms in the literature and therefore look superficially similar. This should be seen as a feature, actually, because simplicity and modularity are important guiding principles in science.

Our contributions are as follows:

- We introduce asymmetric Byzantine quorum systems formally in Section 4 as an extension of standard Byzantine quorum systems and discuss some of their properties.
- In Section 5, we show two implementations of a shared register, with single-writer, multi-reader regular semantics, using asymmetric Byzantine quorum systems.
- We examine broadcast primitives in the Byzantine model with asymmetric trust in Section 6. In particular, we define and implement Byzantine consistent and reliable broadcast protocols. The latter primitive is related to a "federated voting" protocol used by Stellar consensus [22].

The long version of the paper contains more details and all proofs [5].

2 Related work

Damgård et al. [10] introduce some basics of asymmetric trust in the context of synchronous protocols for secure distributed computation by modeling process-specific fail-prone systems. They state the consistency property of asymmetric Byzantine quorums but do not prove that the B^3 property is required.

The *Ripple* consensus protocol is run by an open set of validator nodes. The protocol uses votes, similar to standard consensus protocols, whereby each validator only communicates with the validators in its UNL. Each validator chooses its own UNL, which makes it possible for anyone to participate, in principle, similar to proof-of-work blockchains. Early investigations suggested that the intersection of the UNLs of every two validators should be at least 20% of each list [25], assuming that also less than one fifth of the validators in the UNL of every node might be faulty. An independent analysis by Armknecht et al. [2] later argued that this bound must be more than 40%. A recent technical report of Chase and MacBrough [9, Thm. 8] concludes, under the same assumption of $f < n/5$ faulty nodes in every UNL of size n, that the UNL overlap should actually be at least 90%.

The *Stellar consensus protocol (SCP)* also features open membership and lets every node express its own set of trusted nodes [22]. Generalizing from Ripple's flat lists of unique nodes, every node declares a collection of trusted sets called *quorum slices*, whereby a slice is "the subset of a quorum convincing one particular node of agreement." A *quorum* in Stellar is a set of nodes "sufficient to reach agreement," defined as a set of nodes that contains one slice for each member node. The quorum choices of all nodes together yield a *federated Byzantine quorum systems (FBQS)*. The Stellar white paper states properties of FBQS and protocols that build on them. A recent paper of García-Pérez and Gotsman [12] elaborates on FBQS. Recent papers further describe Stellar's FBQS [18] and a closely related notion called *personal Byzantine quorum systems* [19].

However, these concepts and protocols do not map to known primitives in distributed computing. The FBQS notion is at odds with the usual notion of a Byzantine quorum system in the sense that it does not reduce to a symmetric quorum system for symmetric trust choices.

In contrast to Ripple's and Stellar's attempts to formalize subjective trust, our asymmetric quorum formulation extends the well-established quorum systems that underlie classic

Byzantine consensus. We also introduce several protocols with asymmetric trust that strictly generalize existing standard algorithms, which have so far required common trust and knowledge of all nodes.

3 System model

We consider a system of n *processes* $\mathcal{P} = \{p_1, \ldots, p_n\}$ that communicate with each other. The processes interact asynchronously with each other through exchanging messages. The system itself is asynchronous, i.e., the delivery of messages among processes may be delayed arbitrarily and the processes have no synchronized clocks. Every process is identified by a name, but such identifiers are not made explicit. We use standard notions of protocols, events, functionalities, and fair executions from the literature [20, 4]. All processes are linked by reliable and authenticated point-to-point FIFO channels [14, 4].

A process that follows its protocol during an execution is called *correct*. On the other hand, a *faulty* process may crash or even 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.

Some protocols use *digital signatures*, for which we adopt an idealized implementation with two operations, $sign_i$ and $verify_i$. It follows the standard semantics; details appear in the long version [5].

4 Asymmetric Byzantine quorum systems

4.1 Symmetric trust

Quorum systems are well-known in settings with symmetric trust. As demonstrated by many applications to distributed systems, ordinary quorum systems [23] and Byzantine quorum systems [21] play a crucial role in formulating resilient protocols that tolerate faults through replication [8]. A quorum system typically ensures a consistency property among the processes in an execution, despite the presence of some faulty processes.

For the model with Byzantine faults, *Byzantine quorum systems* have been introduced by Malkhi and Reiter [21]. This notion is defined with respect to a *fail-prone system* $\mathcal{F} \subseteq 2^{\mathcal{P}}$, a collection of subsets of $\mathcal{P}$, 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 [21]. A fail-prone system is the same as the *basis* of an *adversary structure*, which was introduced independently by Hirt and Maurer [15].

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,

▶ **Definition 1** (Byzantine quorum system [21]). *A* 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.,*

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

The above notion is also known as a *Byzantine dissemination quorum system* [21] 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 [24], 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 $S \subseteq T$ [11]. In this sense, a quorum system for $\mathcal{F}$ is *minimal* whenever it does not dominate any other quorum system for $\mathcal{F}$.

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

▶ **Definition 2** (Q^3-condition [21, 15]). *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 $k \geq 2$ [15].

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 Byzantine quorum system.

▶ **Lemma 3** ([21, Theorem 5.4]). *Given a fail-prone system $\mathcal{F}$, a 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 Byzantine quorum system.*

The quorum system $\mathcal{Q} = \overline{\mathcal{F}}$ is called the *canonical quorum system* of $\mathcal{F}$. According to the duality between $\mathcal{Q}$ and $\mathcal{F}$, properties of $\mathcal{F}$ are often ascribed to $\mathcal{Q}$ as well; for instance, we say $Q^3(\mathcal{Q})$ holds if and only if $Q^3(\mathcal{F})$. However, note that the canonical quorum system is not always minimal. For instance, if $\mathcal{F}$ consists of all sets of $f \ll n/3$ processes, then each quorum in the canonical quorum system has $n - f$ members, but also the family of all subsets of $\mathcal{P}$ with $\lceil \frac{n+f+1}{2} \rceil < n - f$ processes forms a quorum system.

Core sets. A *core set* C for $\mathcal{F}$ is a minimal set of processes that contains at least one correct process in every execution. More precisely, $C \subseteq \mathcal{P}$ is a core set whenever (1) for all $F \in \mathcal{F}$, it holds $\mathcal{P} \setminus F \cap C \neq \emptyset$ (and, equivalently, $C \not\subseteq F$) and (2) for all $C' \subsetneq C$, there exists $F \in \mathcal{F}$ such that $\mathcal{P} \setminus F \cap C' = \emptyset$ (and, equivalently, $C' \subseteq F$). With the threshold failure assumption, every set of $f + 1$ processes is a core set. A *core set system* $\mathcal{C}$ is the minimal collection of all core sets, in the sense that no set in $\mathcal{C}$ is contained in another. Core sets can be complemented by *survivor sets*, as shown by Junqueira et al. [16]. This yields a dual characterization of resilient distributed protocols, which parallels ours using fail-prone sets and quorums.

4.2 Asymmetric trust

In our model with asymmetric trust, every process is free to make its own trust assumption and to express this with a fail-prone system. Hence, 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. This notion has earlier been formalized by Damgård et al. [10].

▶ **Definition 4** (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* $\mathcal{Q}_i \subseteq 2^{\mathcal{P}}$ *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.,*

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

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

▶ **Definition 5** (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 is the generalization of Lemma 3 for asymmetric quorum systems; it was stated by Damgård et al. [10] without proof. As for symmetric quorum systems, we use this result and say that $B^3(\mathbb{Q})$ holds whenever the asymmetric $\mathbb{Q}$ consists of the canonical quorum systems for $\mathbb{F}$ and $B^3(\mathbb{F})$ holds. A proof of the following result appears in the long version [5].

▶ **Theorem 6.** *An asymmetric fail-prone system* $\mathbb{F}$ *satisfies* $B^3(\mathbb{F})$ *if and only if there exists an asymmetric quorum system for* $\mathbb{F}$*.*

Kernels. Given a symmetric Byzantine quorum system $\mathcal{Q}$, we define a *kernel* K as a set of processes that overlaps with every quorum and that is minimal in this respect. Formally, $K \subseteq \mathcal{P}$ is a *kernel of* $\mathcal{Q}$ if and only if

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

and

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

The *kernel system* $\mathcal{K}$ of $\mathcal{Q}$ is the set of all kernels of $\mathcal{Q}$.

For example, under a threshold failure assumption where any f processes may fail and the quorums are all sets of $\lceil \frac{n+f+1}{2} \rceil$ processes, every set of $\lfloor \frac{n-f+1}{2} \rfloor$ processes is a kernel.

The definition of a kernel is related to that of a core set in the following sense. For a given maximal fail-prone system $\mathcal{F}$, consider its canonical quorum system $\mathcal{Q} = \overline{\mathcal{F}}$; if $\mathcal{Q}$ is minimal, then the kernel system of $\mathcal{Q}$ is the same as the core-set system for $\mathcal{F}$.

Asymmetric core sets and kernels. Let $\mathbb{F} = [\mathcal{F}_1, \ldots, \mathcal{F}_n]$ be an asymmetric fail-prone system. An *asymmetric core set system* $\mathbb{C}$ is an array of collections of sets $[\mathcal{C}_1, \ldots, \mathcal{C}_n]$ such that each $\mathcal{C}_i$ is a core set system for the fail-prone system $\mathcal{F}_i$. We call a set $C_i \in \mathcal{C}_i$ a *core set for* p_i.

Given an asymmetric 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 a *kernel for* p_i.

Naïve and wise processes. The faults or corruptions occurring in a protocol execution with an underlying quorum system imply a set F of actually *faulty processes*. However, no process knows F and this information is only available to an observer outside the system. With a traditional quorum system $\mathcal{Q}$ designed for a fail-prone set $\mathcal{F}$, the guarantees of a protocol usually hold as long as $F \in \mathcal{F}$. Recall that such protocol properties apply to *correct* processes only but not to faulty ones.

With asymmetric quorums, we further distinguish between two kinds of correct processes, depending on whether they considered F in their trust assumption or not. Given a protocol execution, the processes are therefore partitioned into three 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, since the naïve processes may have the "wrong friends."

Guilds. If too many processes are naïve or even fail during a protocol run with asymmetric quorums, then protocol properties cannot be ensured. A *guild* is a set of wise processes that contains at least one quorum for each member; by definition this quorum consists only of wise processes. A guild ensures liveness and consistency for typical protocols. This generalizes from protocols for symmetric quorum systems, where the correct processes in every execution form a quorum by definition. (A guild represents a group of influential and well-connected wise processes, like in the real world.)

▶ **Definition 7** (Guild). *Given a fail-prone system $\mathbb{F}$, an asymmetric 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:*

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

Superficially a guild seems similar to a "quorum" in the Stellar consensus protocol [22], but the two notions actually differ because a guild contains only wise processes and Stellar's quorums do not distinguish between naïve and wise processes.

Observe that for a specific execution, 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. Hence, every execution with a guild contains a unique *maximal guild* $\mathcal{G}_{\max}$.

Example. We define an example asymmetric fail-prone system $\mathbb{F}_A$ on $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5\}$. The notation $\Theta_k^n(\mathcal{S})$ for a set $\mathcal{S}$ with n elements denotes the "threshold" combination operator and enumerates all subsets of $\mathcal{S}$ of cardinality k. W.l.o.g. every process trusts itself. The diagram below shows fail-prone sets as shaded areas and the notation $\frac{n}{k}$ in front of a fail-prone set stands for k out of the n processes in the set.

$\mathbb{F}_A$:

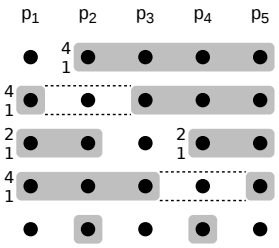
$\mathcal{F}_1 = \Theta_1^4(\{p_2, p_3, p_4, p_5\})$

$\mathcal{F}_2 = \Theta_1^4(\{p_1, p_3, p_4, p_5\})$

$\mathcal{F}_3 = \Theta_1^2(\{p_1, p_2\}) * \Theta_1^2(\{p_4, p_5\})$

$\mathcal{F}_4 = \Theta_1^4(\{p_1, p_2, p_3, p_5\})$

$\mathcal{F}_5 = \{\{p_2, p_4\}\}$



The operator $*$ for two sets satisfies $\mathcal{A} * \mathcal{B} = \{A \cup B | A \in \mathcal{A}, B \in \mathcal{B}\}$. As one can verify in a straightforward way, $B^3(\mathbb{F}_A)$ holds. Let $\mathbb{Q}_A$ be the canonical asymmetric quorum system for $\mathbb{F}_A$. Note that since $\mathbb{F}_A$ contains the fail-prone systems of p_3 and p_5 that permit two faulty processes each, this fail-prone system cannot be obtained as a special case of $\Theta_1^5(\{p_1, p_2, p_3, p_4, p_5\})$. When $F = \{p_2, p_4\}$, for example, then processes p_3 and p_5 are wise and p_1 is naïve.

5 Shared memory

This section illustrates a first application of asymmetric quorum systems: how to emulate shared memory, represented by a *register*. Maintaining a shared register reliably in a distributed system subject to faults is perhaps the most fundamental task for which ordinary, symmetric quorum systems have been introduced, in the models with crashes [13] and with Byzantine faults [21]. We present definitions and one protocol for implementing a register with asymmetric quorums in this section.

The long version [5] also presents a second protocol without digital signatures and explains why federated Byzantine quorum systems according to Stellar [22] fail to directly emulate shared memory. This protocol also illustrates the role of an asymmetric core set system that generalizes the notion of an $(f + 1)$-process subset in the threshold model.

5.1 Definitions

We use the standard notions for operations and their precedence to formalize a register as a shared object. More precisely, a *register* with domain $\mathcal{X}$ provides two operations: *write*(x), which is parameterized by a value $x \in \mathcal{X}$ and outputs a token ACK when it completes; and *read*, which takes no parameter for invocation but outputs a value $x \in \mathcal{X}$ upon completion.

We consider a *single-writer* (or *SW*) register, where only a designated process p_w may invoke *write*, and permit *multiple readers* (or *MR*), that is, every process may execute a *read* operation. The register is initialized with a special value x_0, which is written by an imaginary *write* operation that occurs before any process invokes operations. We consider *regular* semantics under concurrent access [17]; the extension to other forms of concurrent memory, including an atomic register, proceeds analogously.

It is customary in the literature to assume that the writer and reader processes are correct; with asymmetric quorums we assume explicitly that readers and writers are *wise*. We illustrate below why one cannot extend the guarantees of the register to naïve processes.

▶ **Definition 8** (Asymmetric Byzantine SWMR regular register). *A protocol emulating an* asymmetric SWMR regular register *satisfies:*

Liveness: *If a wise process p invokes an operation on the register, p eventually completes the operation.*

Safety: *Every read operation of a wise process that is not concurrent with a write returns the value written by the most recent, preceding write of a wise process; furthermore, a read operation of a wise process concurrent with a write of a wise process may also return the value that is written concurrently.*

5.2 Protocol with authenticated data

In Algorithm 1, we describe a protocol for emulating a regular SWMR register with an asymmetric Byzantine quorum system, for a designated writer p_w and a reader $p_r \in \mathcal{P}$. The protocol uses *data authentication* implemented with digital signatures. This protocol is the same as the classic one of Malkhi and Reiter [21] that uses a Byzantine dissemination quorum system and where processes send messages to each other over point-to-point links. The difference lies in the individual choices of quorums by the processes and that it ensures safety and liveness for wise processes.

In the register emulation, the writer p_w obtains ACK messages from all processes in a quorum $Q_w \in \mathcal{Q}_w$; likewise, the reader p_r waits for a VALUE message carrying a value/-timestamp pair from every process in a quorum $Q_r \in \mathcal{Q}_r$ of the reader.

Algorithm 1 Emulation of an asymmetric SWMR regular register (process p_i).

State
wts: sequence number of write operations, stored only by writer p_w
rid: identifier of read operations, used only by reader
ts, v, σ: current state stored by p_i: timestamp, value, signature

upon invocation *write*(v) **do** // only if p_i is writer p_w
$wts \leftarrow wts + 1$
$\sigma \leftarrow sign_w(\text{WRITE}\|w\|wts\|v)$
send message $[\text{WRITE}, wts, v, \sigma]$ to all $p_j \in \mathcal{P}$
wait for receiving a message [ACK] from all processes in some quorum $Q_w \in \mathcal{Q}_w$

upon invocation *read* **do** // only if p_i is reader p_r
$rid \leftarrow rid + 1$
send message $[\text{READ}, rid]$ to all $p_j \in \mathcal{P}$
wait for receiving messages $[\text{VALUE}, r_j, ts_j, v_j, \sigma_j]$ from all processes in some $Q_r \in \mathcal{Q}_r$ **such that**
$r_j = rid$ **and** $verify_w(\sigma_j, \text{WRITE}\|w\|ts\|v_j)$
return $highestval(\{(ts_j, v_j) | j \in Q_r\})$

upon receiving a message $[\text{WRITE}, ts', v', \sigma']$ from p_w **do** // every process
if $ts' > ts$ **then**
$(ts, v, \sigma) \leftarrow (ts', v', \sigma')$
send message [ACK] to p_w

upon receiving a message $[\text{READ}, r]$ from p_r **do** // every process
send message $[\text{VALUE}, r, ts, v, \sigma]$ to p_r

The function $highestval(S)$ takes a set of timestamp/value pairs S as input and outputs the value in the pair with the largest timestamp, i.e., v such that $(ts, v) \in S$ and $\forall (ts', v') \in S : ts' < ts \vee (ts', v') = (ts, v)$. Note that this v is unique in Algorithm 1 because p_w is correct. The protocol uses digital signatures, modeled by operations $sign_i$ and $verify_i$, as introduced earlier.

▶ **Theorem 9.** *Algorithm 1 emulates an asymmetric Byzantine SWMR regular register.*

Example. We show why the guarantees of this protocol with asymmetric quorums hold only for wise readers and writers. Consider $\mathbb{Q}_A$ from the last section and an execution in which p_2 and p_4 are faulty, and therefore p_1 is naïve and p_3 and p_5 are wise. A quorum for p_1 consists of p_1 and three processes in $\{p_2, \ldots, p_5\}$; moreover, every process set that contains p_3, one of $\{p_1, p_2\}$ and one of $\{p_4, p_5\}$ is a quorum for p_3.

We illustrate that if naïve p_1 writes, then a wise reader p_3 may violate safety. Suppose that all correct processes, especially p_3, store timestamp/value/signature triples from an operation that has terminated and that wrote x. When p_1 invokes *write*(u), it obtains [ACK] messages from all processes except p_3. This is a quorum for p_1. Then p_3 runs a *read* operation and receives the outdated values representing x from itself (p_3 is correct but has not been involved in writing u) and also from the faulty p_2 and p_4. Hence, p_3 outputs x instead of u.

Analogously, with the same setup of every process initially storing a representation of x but with wise p_3 as writer, suppose p_3 executes *write*(u). It obtains [ACK] messages from p_2, p_3, and p_4 and terminates. When p_1 subsequently invokes *read* and receives values representing x, from correct p_1 and p_5 and from faulty p_2 and p_4, then p_1 outputs x instead of y and violates safety as a naïve reader.

Since the sample operations are not concurrent, the implication actually holds also for registers with only safe semantics.

6 Broadcast

This section shows how to implement two *broadcast primitives* tolerating Byzantine faults with asymmetric quorums. Recall from the standard literature [14, 8, 4] that reliable broadcasts offer basic forms of reliable message delivery and consistency, but they do not impose a total order on delivered messages (as this is equivalent to consensus). The Byzantine broadcast primitives described here, *consistent broadcast* and *reliable broadcast*, are prominent building blocks for many more advanced protocols.

With both primitives, the sender process may broadcast a message m by invoking *broadcast*(m); the broadcast abstraction outputs m to the local application on the process through a *deliver*(m) event. Moreover, the notions of broadcast considered in this section are intended to deliver only one message per instance. Every instance has a distinct (implicit) label and a designated sender p_s. With standard multiplexing techniques one can extend this to a protocol in which all processes may broadcast messages repeatedly [4].

Byzantine consistent broadcast. The simplest such primitive, which has been called *(Byzantine) consistent broadcast* [4], ensures only that those correct processes which deliver a message agree on the content of the message, but they may not agree on termination. In other words, the primitive does not enforce "reliability" such that a correct process outputs a message if and only if all other correct processes produce an output. The events in its interface are denoted by *c-broadcast* and *c-deliver*.

The change of the definition towards asymmetric quorums affects most of its guarantees, which hold only for wise processes but not for all correct ones. This is similar to the definition of a register in Section 5.

▶ **Definition 11** (Asymmetric Byzantine consistent broadcast)**.** *A protocol for* asymmetric (Byzantine) consistent broadcast *satisfies:*

Validity: *If a correct process p_s c-broadcasts a message m, then all wise processes eventually c-deliver m.*

Algorithm 2 Asymmetric Byzantine consistent broadcast protocol with sender p_s (process p_i).

State
sentecho ← FALSE: indicates whether p_i has sent ECHO
echos ← $[\perp]^N$: collects the received ECHO messages from other processes
delivered ← FALSE: indicates whether p_i has delivered a message

upon invocation *c-broadcast*(m) **do**
send message $[\text{SEND}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{SEND}, m]$ from p_s **such that** $\neg$*sentecho* **do**
sentecho ← TRUE
send message $[\text{ECHO}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{ECHO}, m]$ from p_j **do**
if $echos[j] = \perp$ **then**
$echos[j] \leftarrow m$

upon exists $m \neq \perp$ **such that** $\{p_j \in \mathcal{P} | echos[j] = m\} \in \mathcal{Q}_i$ **and** $\neg$*delivered* **do**
delivered ← TRUE
output *c-deliver*(m)

Consistency: *If some wise process c-delivers m and another wise process c-delivers m', then $m = m'$.*

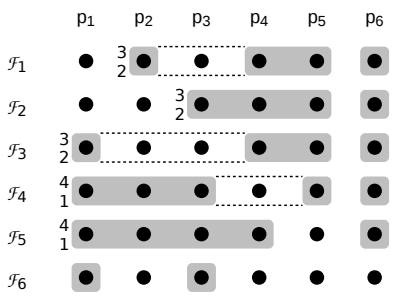
Integrity: *For any message m, every correct process c-delivers m at most once. Moreover, if the sender p_s is correct and the receiver is wise, then m was previously c-broadcast by p_s.*

The following protocol is an extension of "authenticated echo broadcast" [4], which goes back to Srikanth and Toueg [26]. It is a building block found in many Byzantine fault-tolerant protocols with greater complexity. The adaptation for asymmetric quorums is straightforward: Every process considers its own quorums before *c-delivering* the message.

▶ **Theorem 12.** *Algorithm 2 implements asymmetric Byzantine consistent broadcast.*

Example. We illustrate the broadcast protocols using a six-process asymmetric quorum system $\mathbb{Q}_B$, defined through its fail-prone system $\mathbb{F}_B$. In $\mathbb{F}_B$, as shown below, for p_1, p_2, and p_3, each process always trusts itself, some other process of $\{p_1, p_2, p_3\}$ and one further process in $\{p_1, \ldots, p_5\}$. Process p_4 and p_5 each assumes that at most one other process of $\{p_1, \ldots, p_5\}$ may fail (excluding itself). Moreover, none of the processes p_1, …, p_5 ever trusts p_6. For p_6 itself, the fail-prone set is $\{p_1, p_3\}$, i.e., it trusts p_2, p_4, and p_5 unconditionally.

$$\mathbb{F}_B\text{:}\quad \begin{aligned}
\mathcal{F}_1 &= \Theta_2^3(\{p_2, p_4, p_5\}) * \{\{p_6\}\} \\
\mathcal{F}_2 &= \Theta_2^3(\{p_3, p_4, p_5\}) * \{\{p_6\}\} \\
\mathcal{F}_3 &= \Theta_2^3(\{p_1, p_4, p_5\}) * \{\{p_6\}\} \\
\mathcal{F}_4 &= \Theta_1^4(\{p_1, p_2, p_3, p_5\}) * \{\{p_6\}\} \\
\mathcal{F}_5 &= \Theta_1^4(\{p_1, p_2, p_3, p_4\}) * \{\{p_6\}\} \\
\mathcal{F}_6 &= \{\{p_1, p_3\}\}
\end{aligned}$$



One can verify that $B^3(\mathbb{F}_B)$ holds; hence, let $\mathbb{Q}_B$ be the canonical quorum system of $\mathbb{F}_B$. Again, there is no reliable process that could be trusted by all and $\mathbb{Q}_B$ is not a special case of a symmetric threshold Byzantine quorum system. With $F = \{p_1, p_5\}$, for instance, process p_3 is wise, p_2, p_4, and p_6 are naïve, and there is no guild.

Consider now an execution of Algorithm 2 with sender p_4^* and $F = \{p_4^*, p_5^*\}$ (we write p_4^* and p_5^* to denote that they are faulty). This means processes p_1, p_2, p_3 are wise and form a guild because $\{p_1, p_2, p_3\}$ is a quorum for all three; furthermore, p_6 is naïve. The protocol execution proceeds as follows.

$$p_4^* : \begin{cases} [\text{SEND}, x] \to p_1, p_3 \\ [\text{SEND}, u] \to p_2, p_6 \end{cases}$$

$$\begin{aligned} p_1 &: \ [\text{ECHO}, x] \to \mathcal{P} && p_1 : \textit{c-deliver}(x) \\ p_2 &: \ [\text{ECHO}, u] \to \mathcal{P} && p_2 : \text{no quorum of } [\text{ECHO}] \text{ in } \mathcal{Q}_2 \\ p_3 &: \ [\text{ECHO}, x] \to \mathcal{P} && p_3 : \text{no quorum of } [\text{ECHO}] \text{ in } \mathcal{Q}_3 \end{aligned}$$

$$p_4^* : \begin{cases} [\text{ECHO}, x] \to p_1 \\ [\text{ECHO}, u] \to p_6 \end{cases} \qquad p_5^* : \begin{cases} [\text{ECHO}, x] \to p_1 \\ [\text{ECHO}, u] \to p_6 \end{cases}$$

$$p_6 : \ [\text{ECHO}, u] \to \mathcal{P} \qquad p_6 : \textit{c-deliver}(u)$$

Hence, p_1 receives $[\text{ECHO}, x]$ from, say, $\{p_1, p_3, p_4^*\} \in \mathcal{Q}_1$ and *c-delivers* x, but the other wise processes do not terminate. The naïve p_6 gets $[\text{ECHO}, u]$ from $\{p_2, p_4^*, p_5^*, p_6\} \in \mathcal{Q}_6$ and *c-delivers* $u \neq x$.

Byzantine reliable broadcast. In the symmetric setting, consistent broadcast has been extended to *(Byzantine) reliable broadcast* in a well-known way to address the disagreement about termination among the correct processes [4]. This primitive has the same interface as consistent broadcast, except that its events are called *r-broadcast* and *r-deliver* instead of *c-broadcast* and *c-deliver*, respectively.

A reliable broadcast protocol also has all properties of consistent broadcast, but satisfies the additional *totality* property stated next. Taken together, *consistency* and *totality* imply a notion of *agreement*, similar to what is also ensured by many crash-tolerant broadcast primitives. Analogously to the earlier primitives with asymmetric trust, our notion of an *asymmetric reliable broadcast*, defined next, ensures agreement on termination only for the wise processes, and moreover only for executions with a guild. Also the *validity* of Definition 11 is extended by the assumption of a guild. Intuitively, one needs a guild because the wise processes that make up the guild are self-sufficient, in the sense that the guild contains a quorum of wise processes for each of its members; without that, there may not be enough wise processes.

▶ **Definition 13** (Asymmetric Byzantine reliable broadcast)**.** *A protocol for* asymmetric (Byzantine) reliable broadcast *is a protocol for asymmetric Byzantine consistent broadcast with the revised* validity *condition and the additional* totality *condition stated next:*

Validity: *In all executions with a guild, if a correct process p_s c-broadcasts a message m, then all processes in the maximal guild eventually c-deliver m.*

Totality: *In all executions with a guild, if a wise process r-delivers some message, then all processes in the maximal guild eventually r-deliver a message.*

The protocol of Bracha [3] implements reliable broadcast subject to Byzantine faults with symmetric trust. It augments the authenticated echo broadcast from Algorithm 2 with a second all-to-all exchange, where each process is supposed to send READY with the

payload message that will be *r-delivered*. When a process receives the same m in $2f+1$ READY messages, in the symmetric model with a threshold Byzantine quorum system, then it *r-delivers* m. Also, a process that receives $[\text{READY}, m]$ from $f+1$ distinct processes and that has not yet sent a READY chimes in and also sends $[\text{READY}, m]$. These two steps ensure totality.

For asymmetric quorums, the conditions of a process p_i receiving $f+1$ and $2f+1$ equal READY messages, respectively, generalize to receiving the same message from a kernel for p_i and from a quorum for p_i. Intuitively, the change in the first condition ensures that when a wise process p_i receives the same $[\text{READY}, m]$ message from a kernel for itself, then this kernel intersects with some quorum of wise processes. Therefore, at least one wise process has sent $[\text{READY}, m]$ and p_i can safely adopt m. Furthermore, the change in the second condition relies on the properties of asymmetric quorums to guarantee that whenever some wise process has *r-delivered* m, then enough correct processes have sent a $[\text{READY}, m]$ message such that all wise processes eventually receive a kernel of $[\text{READY}, m]$ messages and also send $[\text{READY}, m]$.

Applying these changes to Bracha's protocol results in the asymmetric reliable broadcast protocol shown in Algorithm 3. Note that it strictly extends Algorithm 2 by the additional round of READY messages, in the same way as for symmetric trust. For instance, when instantiated with the symmetric threshold quorum system of $n = 3f+1$ processes, of which f may fail, then every set of $f+1$ processes is a kernel.

In Algorithm 3, there are two conditions that let a correct p_i send $[\text{READY}, m]$: either receiving a quorum of $[\text{ECHO}, m]$ messages for itself or obtaining a kernel for itself of $[\text{READY}, m]$ messages. For the first case, we say p_i *sends* READY *after* ECHO; for the second case, we say p_i *sends* READY *after* READY.

▶ **Lemma 14.** *In any execution with a guild, there exists a unique m such that whenever a wise process sends a* READY *message, it contains m.*

This lemma follows from the fact that Algorithm 3 extends Algorithm 2 for consistent broadcast, combined with the consistency property in Definition 11. This shows why the lemma holds for READY messages sent by wise processes after ECHO. For READY messages sent after READY, a new argument is needed, which relies on the properties of kernels and appears in the long version [5].

▶ **Theorem 15.** *Algorithm 3 implements asymmetric Byzantine reliable broadcast.*

Example. Consider again the protocol execution with $\mathbb{Q}_B$ introduced earlier for illustrating asymmetric consistent broadcast. Recall that $F = \{p_4^*, p_5^*\}$, the set $\{p_1, p_2, p_3\}$ is a guild, and p_6 is naïve. The start of the execution is the same as shown previously and omitted. Instead of *c-delivering* x and u, respectively, p_1 and p_6 send $[\text{READY}, x]$ and $[\text{READY}, u]$ to all processes:

...	$p_1 : [\text{READY}, x] \to \mathcal{P}$			p_1 : *r-deliver*(x)
...	p_2 : no quorum	$p_2 : [\text{READY}, x] \to \mathcal{P}$		p_2 : *r-deliver*(x)
...	p_3 : no quorum		$p_3 : [\text{READY}, x] \to \mathcal{P}$	p_3 : *r-deliver*(x)
...	p_4^* : –			
...	p_5^* : –			
...	$p_6 : [\text{READY}, u] \to \mathcal{P}$	p_6 : no quorum		

Algorithm 3 Asymmetric Byzantine reliable broadcast protocol with sender p_s (process p_i).

State
- *sentecho* $\leftarrow$ FALSE: indicates whether p_i has sent ECHO
- *echos* $\leftarrow [\bot]^N$: collects the received ECHO messages from other processes
- *sentready* $\leftarrow$ FALSE: indicates whether p_i has sent READY
- *readys* $\leftarrow [\bot]^N$: collects the received READY messages from other processes
- *delivered* $\leftarrow$ FALSE: indicates whether p_i has delivered a message

upon invocation *r-broadcast*(m) **do**
- send message $[\text{SEND}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{SEND}, m]$ from p_s **such that** $\neg$*sentecho* **do**
- *sentecho* $\leftarrow$ TRUE
- send message $[\text{ECHO}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{ECHO}, m]$ from p_j **do**
- **if** *echos*$[j] = \bot$ **then**
  - *echos*$[j] \leftarrow m$

upon exists $m \neq \bot$ **such that** $\{p_j \in \mathcal{P} | \textit{echos}[j] = m\} \in \mathcal{Q}_i$ **and** $\neg$*sentready* **do**
- // a quorum for p_i
- *sentready* $\leftarrow$ TRUE
- send message $[\text{READY}, m]$ to all $p_j \in \mathcal{P}$

upon exists $m \neq \bot$ **such that** $\{p_j \in \mathcal{P} | \textit{readys}[j] = m\} \in \mathcal{K}_i$ **and** $\neg$*sentready* **do**
- // a kernel for p_i
- *sentready* $\leftarrow$ TRUE
- send message $[\text{READY}, m]$ to all $p_j \in \mathcal{P}$

upon receiving a message $[\text{READY}, m]$ from p_j **do**
- **if** *readys*$[j] = \bot$ **then**
  - *readys*$[j] \leftarrow m$

upon exists $m \neq \bot$ **such that** $\{p_j \in \mathcal{P} | \textit{readys}[j] = m\} \in \mathcal{Q}_i$ **and** $\neg$*delivered* **do**
- *delivered* $\leftarrow$ TRUE
- **output** *r-deliver(m)*

Note that the kernel systems of processes p_1, p_2, and p_3 are, respectively, $\mathcal{K}_1 = \{\{p_1\}, \{p_3\}\}$, $\mathcal{K}_2 = \{\{p_1\}, \{p_2\}\}$, and $\mathcal{K}_3 = \{\{p_2\}, \{p_3\}\}$. Hence, when p_2 receives $[\text{READY}, x]$ from p_1, it sends $[\text{READY}, x]$ in turn because $\{p_1\}$ is a kernel for p_2, and when p_3 receives this message, then it sends $[\text{READY}, x]$ because $\{p_2\}$ is a kernel for p_3.

Furthermore, since $\{p_1, p_2, p_3\}$ is the maximal guild and contains a quorum for each of its members, all three wise processes *r-deliver* x as implied by *consistency* and *totality*. The naïve p_6 does not *r-deliver* anything, however.

Remarks. Asymmetric reliable broadcast (Definition 13) ensures validity and totality only for processes in the maximal guild. On the other hand, an asymmetric consistent broadcast (Definition 11) ensures validity also for all *wise* processes. We leave it as an open problem to determine whether these guarantees can also be extended to wise processes for asymmetric reliable broadcast and the Bracha protocol. This question is equivalent to determining whether there exist any wise processes outside the maximal guild.

Another open problem concerns the conditions for reacting to READY messages in the asymmetric reliable broadcast protocol. Already in Bracha's protocol for the threshold model [3], a process (1) sends its own READY message upon receiving $f + 1$ READY messages and (2) *r-delivers* an output upon receiving $2f + 1$ READY messages. These conditions generalize for arbitrary, non-threshold quorum systems to receiving messages (1) from any set that is guaranteed to contain at least one correct process and (2) from any set that still contains at least one process even when any two fail-prone process sets are subtracted. In Algorithm 3, in contrast, a process delivers the payload only after receiving READY messages from one of its quorums. But such a quorum (e.g., $\lceil \frac{n+f+1}{2} \rceil$ processes) may be larger than a set in the second case (e.g., $2f + 1$ processes). It remains interesting to find out whether this discrepancy is necessary.

References

1. Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed, and Jason Yellick. Hyperledger Fabric: A distributed operating system for permissioned blockchains. In *Proc. 13th European Conference on Computer Systems (EuroSys)*, pages 30:1–30:15, April 2018. doi:10.1145/3190508.3190538.
2. Frederik Armknecht, Ghassan O. Karame, Avikarsha Mandal, Franck Youssef, and Erik Zenner. Ripple: Overview and outlook. In Mauro Conti, Matthias Schunter, and Ioannis G. Askoxylakis, editors, *Proc. Trust and Trustworthy Computing (TRUST)*, volume 9229 of *Lecture Notes in Computer Science*, pages 163–180. Springer, 2015.
3. Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75:130–143, 1987.
4. Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming (Second Edition)*. Springer, 2011.
5. Christian Cachin and Björn Tackmann. Asymmetric distributed trust. e-print, arXiv:1906.09314 [cs.DC], 2019. URL: http://arxiv.org/abs/1906.09314.
6. Christian Cachin and Marko Vukolić. Blockchain consensus protocols in the wild. In Andréa W. Richa, editor, *Proc. 31st Intl. Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 1:1–1:16, 2017. doi:10.4230/LIPIcs.DISC.2017.1.
7. Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.
8. Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors. *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*. Springer, 2010.
9. Brad Chase and Ethan MacBrough. Analysis of the XRP ledger consensus protocol. e-print, arXiv:1802.07242 [cs.DC], 2018.
10. Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In Kaoru Kurosawa, editor, *Advances in Cryptology: ASIACRYPT 2007*, volume 4833 of *Lecture Notes in Computer Science*. Springer, 2007.
11. Hector Garcia-Molina and Daniel Barbara. How to assign votes in a distributed system. *jacm*, 32(4):841–860, 1985.
12. Álvaro García-Pérez and Alexey Gotsman. Federated Byzantine quorum systems. In *Proc. 22nd Conference on Principles of Distributed Systems (OPODIS)*, volume 125 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17:1–17:16, 2018. doi:10.4230/LIPIcs.OPODIS.2018.17.
13. David K. Gifford. Weighted voting for replicated data. In *Proc. 7th ACM Symposium on Operating System Principles (SOSP)*, pages 150–162, 1979.

14 Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems (2nd Ed.)*. ACM Press & Addison-Wesley, New York, 1993.

15 Martin Hirt and Ueli Maurer. Player simulation and general adversary structures in perfect multi-party computation. *Journal of Cryptology*, 13(1):31–60, 2000.

16 Flavio P. Junqueira, Keith Marzullo, Maurice Herlihy, and Lucia Draque Penso. Threshold protocols in survivor set systems. *Distributed Computing*, 23:135–149, 2010.

17 Leslie Lamport. On interprocess communication. *Distributed Computing*, 1(2):77–85, 86–101, 1986.

18 Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafal Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proc. 27th ACM Symposium on Operating Systems Principles (SOSP)*, pages 80–96, 2019.

19 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In Jukka Suomela, editor, *Proc. 33rd International Symposium on Distributed Computing (DISC 2019)*, volume 146 of *LIPIcs*, pages 27:1–27:15, 2019. doi:10.4230/LIPIcs.DISC.2019.27.

20 Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, San Francisco, 1996.

21 Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

22 David Mazières. The Stellar consensus protocol: A federated model for Internet-level consensus. Stellar, available online, https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2016.

23 Moni Naor and Avishai Wool. The load, capacity and availability of quorum systems. *SIAM Journal on Computing*, 27(2):423–447, 1998.

24 M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

25 David Schwartz, Noah Youngs, and Arthur Britto. The Ripple protocol consensus algorithm. Ripple Labs, available online, https://ripple.com/files/ripple_consensus_whitepaper.pdf, 2014.

26 T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2:80–94, 1987.