Distributed Computing (2026) 39:16
https://doi.org/10.1007/s00446-026-00507-0

Satrapy: From abstract to practical consensus for heterogeneous quorum systems

Xiao Li[1] · **Eric M. Chan**[1] · **Mohsen Lesani**[2]

Received: 19 February 2024 / Accepted: 20 December 2025 / Published online: 20 May 2026

Abstract

The traditional Byzantine quorum-system model assumes a pre-existing, global agreement on the set of quorums (typically defined as the sets consisting of more than two-thirds of the participants). This assumption is problematic in permissionless systems, which strive to allow anyone to join or leave the system dynamically. While proof-of-stake permissionless systems like Ethereum require newly joining participants to register into the system, other permissionless systems like the Ripple Ledger or the Stellar network allow participants to join the system without synchronization by forgoing agreement on the set of quorums. This results in what we call a heterogeneous quorum system, where each participant has its own, personal set of quorums. An important question is to determine under what condition is it possible to solve synchronization problems like reliable broadcast or consensus in a heterogeneous quorum system. In this work, we show that the traditional quorum intersection and quorum availability conditions are not sufficient in heterogeneous quorum systems. Moreover, we propose quorum subsumption, a new condition which, together with quorum availability and quorum intersection, is sufficient to allow solving reliable broadcast and consensus. Finally, we propose protocols for reliable broadcast and consensus in heterogeneous quorum systems that satisfy quorum subsumption. In particular, we present a practical consensus protocol called Satrapy which in contrast to abstract consensus protocols uses finite state and messages.

Keywords Distributed Systems · Impossibility Results · Byzantine fault tolerance

Contents

✉ Xiao Li
xli289@ucr.edu

Eric M. Chan
eric.chan008@email.ucr.edu

Mohsen Lesani
mlesani@ucsc.edu

[1] BCOE, University of California, Riverside, 900 University Ave., Riverside 92521, California, USA

[2] BSE, University of California, Santa Cruz, 1156 High Street, Santa Cruz 95064, California, USA

1 Introduction

Bitcoin [42] had the promise to democratize the global finance. Globally scattered servers validate and process transactions, and maintain a consistent replication of a ledger. However, the nature of the proof-of-work consensus exhibited disadvantages such as high energy consumption and low throughput. In contrast, Byzantine replication has always had modest energy consumption. Further, since its advent as PBFT [18], many recent extensions [6, 12, 13, 17, 39, 47, 48] have improved its throughput. However, its basic model

of quorums is closed and homogeneous: the set of processes are fixed, and the quorums are assumed to be uniform across processes. Thus, projects such as Ripple [44] and Stellar [33, 38] emerged to bring heterogeneity and openness to Byzantine quorum systems. They let processes declare their own set of quorums, or the processes they trust, called slices, from which quorums are calculated.

In this paper, we first consider a basic model of heterogeneous quorum systems where each process has an individual set of quorums. Then, we consider fundamental questions about their properties. Quorum systems are the foundation of common distributed computing abstractions such as reliable broadcast and consensus. We specify the expected safety and liveness properties for these abstractions. What are the necessary and sufficient properties of heterogeneous quorum systems to support these abstractions? Previous work [34] noted that the *quorum intersection* and *weak availability* properties are necessary for the quorum system to implement the consensus abstraction. Quorum intersection requires every pair of quorums to overlap at a well-behaved process. The safety of consensus relies on the quorum intersection of the underlying quorum system: intuitively, if an operation communicates with a quorum, and a different operation communicates with another quorum, a single well-behaved process in their intersection makes the second quorum aware of the first. A quorum system is weakly available for a process if it has a quorum for that process whose members are all well-behaved. Intuitively, for liveness, since a process must communicate with at least one quorum to terminate, the quorum system is available to that process through that quorum.

The quorum intersection and availability properties are necessary. Are they sufficient as well? In this paper, we prove that they are not sufficient conditions to implement reliable broadcast and consensus. For each abstraction, we present execution scenarios, and apply indistinguishability arguments to show that any protocol violates at least one of the specifications. What property should be added to make the properties sufficient? A less known property is quorum sharing [34]. Roughly speaking, every quorum should include a quorum for all its members. This is a property that trivially holds for homogeneous quorum systems, where quorums are uniform across its members. However, this generally does not hold for heterogeneous quorum systems.

Since the quorums of Byzantine processes are arbitrary, in practice, quorum sharing is too strong of a requirement. In order to require inclusion only for the quorums of a well-behaved subset of processes, we consider a weaker notion, called *quorum subsumption*. As we will see, this property lets processes in the included quorum make local decisions while preserving the properties of the including quorum. We precisely capture this property, and show that together with the other two properties, it is sufficient to implement reliable broadcast and consensus abstractions. We present protocols for both reliable broadcast and consensus, and prove that if the underlying quorum system has quorum intersection, availability, and subsumption for certain quorums, then the protocols satisfy the required safety and liveness specifications. We also present bunched voting and practical consensus protocols called Satrapy[1] that use finite state and only send and receive finite number of messages. They are a refinement of the reliable broadcast and consensus protocols mentioned before in order to be amendable for implementation.

In summary, this paper makes the following contributions.

- Properties of quorum-based protocols (§3) and specifications of reliable broadcast and consensus on heterogeneous quorum systems (§4).
- Proof of insufficiency of quorum intersection and availability to solve consensus (§5.1) and reliable broadcast (§5.2).
- Sufficiency of quorum intersection, availability and subsumption to solve reliable broadcast and consensus. We present protocols for reliable broadcast (§6.1) and consensus (§6.2), and their proofs of correctness.
- We present protocols (§7) to solve consensus in practical systems with finite states and messages transmitted.

2 Heterogeneous quorum systems

A quorum is a subset of processes that are collectively trusted to perform an operation. However, this trust may not be uniform: while a process may trust a part of a system, another process may not trust that same part. In this section, we adopt a general model of quorum systems [32, 34] and its properties. These basic definitions adapt common properties of quorum systems to the heterogeneous setting, and serve as the foundation for the theorems and protocols in later sections. Since we want the theorems to be as strong as possible, we introduce the weak notion of quorum subsumption in this paper.

2.1 Processes and quorums

Processes and Failures. We consider a set of processes $\mathcal{P}$ partitioned into a set of *well-behaved* processes $\mathcal{W}$ and a set of *Byzantine* processes $\mathcal{B}$. Well-behaved processes follow the given protocol, while Byzantine processes can deviate from the protocol arbitrarily.

[1] In the Ancient Persian Empire's federated structure, satrapies were heterogeneous semi-autonomous provinces.

Processes communicate by sending each other messages using point-to-point links. We assume that there is a point-to-point link between each pair of processes. A point-to-point link provides the following guarantees: (1) every message sent by a well-behaved process to another well-behaved process is eventually delivered by the latter, (2) no message is delivered to a process more than once, and (3) no message is delivered from a process unless it was sent by that process.

We assume that the network is partially synchronous [20], *i.e.*, that there is an unknown global stabilization time (noted GST) and a known delay such that, after GST, every message sent by a well-behaved process to a well-behaved process is received within this time. Before GST, messages may be arbitrarily delayed and lost. We also assume that processes have local clocks that tick at the same rate.

Heterogeneous Quorum Systems (HQS). In a heterogeneous quorum system, each process has its own, individual set of quorums. An *individual quorum* q of a process p is a minimal, non-empty subset of processes in $\mathcal{P}$ that p trusts to collectively perform an operation. Any superset of a quorum of p is also a quorum of p. For example, let the individual quorums of process 1 be $\{\{1, 2\}, \{1, 3\}\}$. Then, the set $\{1, 2, 4\}$ is a quorum of 1 but is not an individual quorum of 1 as it is not minimal. A process only needs to keep its individual quorums. Naturally, every quorum of a process p contains p itself. (Though, this assumption is not necessary for any of the results in this paper.)

Definition 1 *(Heterogeneous Quorum System)* A heterogeneous quorum system $\mathcal{Q}$ is a mapping from well-behaved processes to their non-empty set of individual quorums.

Fig. 1 presents an example quorum system. Note that when the set of Byzantine processes is known, we omit specifying the quorums of Byzantine processes, since their behavior is arbitrary regardless of any quorum they may have.

When obvious from the context, we use $\mathcal{Q}$ to refer to the set of all individual quorums of the system, *i.e.*, the union of all sets in the co-domain of $\mathcal{Q}$. Additionally, we say quorum systems to refer to heterogeneous quorum systems.

In dissemination quorum systems (DQS) [37] (and the cardinality-based quorum systems as a special case), quorums are uniform for all processes, *i.e.*, processes have the same set of individual quorums. For example, a quorum system that tolerates f Byzantine failures out of $3f+1$ processes considers any set of $2f+1$ processes as a quorum for all processes.

Definition 2 *(Follower)* Process p is a *follower* of process p' if there exists a quorum $q \in \mathcal{Q}(p)$ that includes p'.

$$\begin{aligned}\mathcal{P} &= \mathcal{W} \cup \mathcal{B}, \quad \mathcal{W} = \{1, 3, 4, 5\}, \quad \mathcal{B} = \{2\}\\ \mathcal{Q} &= \{1 \mapsto \{\{1, 2, 3\}, \{1, 4\}\},\\ &\qquad 3 \mapsto \{\{3, 4\}, \{1, 3\}\},\\ &\qquad 4 \mapsto \{\{3, 4\}\},\\ &\qquad 5 \mapsto \{\{1, 2, 3, 5\}\}\}\end{aligned}$$

Fig. 1 Quorum System Example

2.2 Properties

A quorum system is expected to maintain certain properties in order to provide distributed abstractions such as Byzantine reliable broadcast and consensus. Quorum intersection and quorum availability are well-established requirements for quorum systems. In the following section, we will see their adaption to HQS. Further, we identify a new property we call quorum subsumption that helps achieve the aforementioned abstractions in HQS. Finally, we briefly present a few related quorum systems and their properties.

Quorum Intersection. Processes store and retrieve information from the quorum system by communicating with its quorums. To ensure that information is properly passed from one quorum to another, the quorum system is expected to maintain a well-behaved process at the intersection of every pair of quorums. For example, in Fig. 1, all the quorums of well-behaved processes intersect at at least one of well-behaved processes in $\{1, 3, 4\}$.

Definition 3 *(Quorum Intersection)* A quorum system $\mathcal{Q}$ has *quorum intersection* if every pair of quorums of well-behaved processes in $\mathcal{Q}$ intersect at a well-behaved process, *i.e.*, $\forall p, p' \in \mathcal{W}.\ q \in \mathcal{Q}(p).\ q' \in \mathcal{Q}(p').\ q \cap q' \cap \mathcal{W} \neq \emptyset$.

Quorum Availability. In order to support progress for a process, the quorum system is expected to have at least one quorum for that process whose members are all well-behaved. In which we say the quorum system is weakly available for that process. (In the literature, this notion of availability is often unqualified, but we explicitly contrast the weak notion to the strong notion that we define later.) In classical quorum systems, any quorum is a quorum for all processes. This guarantees that if the quorum system is available for a process, it is available for all processes. However, this is obviously not true in a heterogeneous quorum system where quorums are not uniform. In this setting, we weaken the availability property so that only a subset of well-behaved processes have a well-behaved quorum, instead of all well-behaved processes. In Fig. 1, $\mathcal{Q}$ is available for the set $\{1, 3, 4\}$: the quorum $\{1, 4\}$ of process 1, and the quorum $\{3, 4\}$ of processes 3 and 4 make them weakly available. If a quorum system is available, at least one well-behaved process can communicate with a quorum independent of Byzantine processes.

Definition 4 *(Weak Availability)* A quorum system $\mathcal{Q}$ is *weakly available* for a set of processes P if every process in P has at least one quorum that is a subset of $\mathcal{W}$, the set of well-behaved processes. A quorum system $\mathcal{Q}$ is *available* if it is available for a non-empty set of processes.

With quorum availability introduced, we can consider when a quorum system is unavailable. A quorum system is unavailable for a process when that process has no quorum in $\mathcal{W}$, *i.e.*, the Byzantine processes $\mathcal{B}$ can block every one of its quorums. We generalize this idea in the notion of blocking.

Definition 5 *(Blocking Set)* A set of processes P is a blocking set for a process p (or is p-blocking) if P intersects every quorum of p.

For example, consider a cardinality-based quorum system where the system contains $3f+1$ processes and a quorum is any set of $2f+1$ processes. Then, any set of $f+1$ processes is a blocking set for all well-behaved processes, since a set with $f+1$ intersects every quorum. In Fig. 1, process 5 is blocked by $\{2\}$, since its only quorum $\{1, 2, 3, 5\}$ intersect with $\{2\}$.

Notice also that the definition does not stipulate that the blocking set is Byzantine, but rather it is more general. The concept of blocking will be useful for designing our protocols in (§3). For now, we prove a lemma for blocking sets. In order to state the lemma, we generalize the notion of weak availability. We say that a quorum system is weakly available for a set of processes P at a subset of well-behaved processes $W \subseteq \mathcal{W}$ if every process in P has at least one quorum in W. (In Definition 4, W is taken to be the entire set $\mathcal{W}$.)

Lemma 1 *If a quorum system $\mathcal{Q}$ is weakly available for a set of processes P at $W \subseteq \mathcal{W}$, every blocking set of every process in P intersects W.*

Proof Consider a quorum system that is weakly available for P at W, some process p in P, and a set of processes B that blocks p. Then, at least one quorum q of p is a subset of W. By definition of blocking set, B intersects with q. Hence, B intersects W as well. □

Letting W be $\mathcal{W}$ simply results in the following corollary.

Corollary 1 *In every quorum system that is weakly available for a set of processes P, every blocking set of every process in P has at least one well-behaved process.*

Quorum subsumption. We now introduce the notion of quorum subsumption.

Definition 6 *(Quorum Subsumption)* A quorum system $\mathcal{Q}$ is quorum subsuming for a quorum q if every process in q has a quorum that is included in q, *i.e.*, $\ell p \in q.\ \exists q' \in \mathcal{Q}(p).\ q' \subseteq q$. We say $\mathcal{Q}$ is quorum subsuming for a set of quorums if it is subsuming for every quorum in the set.

In Fig. 1, $\mathcal{Q}$ is quorum subsuming for $\{3, 4\}$: both members of this quorum have $\{3, 4\}$ as a quorum, which is a subset of itself. However, $\mathcal{Q}$ is not subsuming for $\{1, 4\}$: the only quorum of process 4 is $\{3, 4\}$, which is not a subset of $\{1, 4\}$. We note that since a quorum system $\mathcal{Q}$ is a mapping from well-behaved processes, it can be quorum subsuming for a quorum q only if q is a subset of well-behaved processes.

Quorum subsumption is inspired by and weakens the notion of *quorum sharing* [34]. Quorum sharing requires the above subsumption property for all quorums.

Definition 7 *(Quorum Sharing)* A quorum system $\mathcal{Q}$ has *quorum sharing* if every process in every quorum q of $\mathcal{Q}$ has a quorum that is included in q, *i.e.*, $\forall q \in \mathcal{Q}.\ \forall p \in q.\ \exists q' \in \mathcal{Q}(p).\ q' \subseteq q$.

However, no property can be required from quorums of Byzantine processes. A property can be required only for quorums of a well-behaved subset of processes. Therefore, we define the weaker notion of quorum subsumption for a subset of quorums, and later show that it is sufficient to implement broadcast and consensus. Quorum subsumption supports both safety and liveness properties of these protocols.

Complete Quorum. We will later see that quorum availability and quorum subsumption are important together for liveness. We succinctly combine the two properties into the notion of complete quorums.

Definition 8 *(Complete Quorum)* A quorum q in a quorum system $\mathcal{Q}$ is a *complete quorum* if all its members are well-behaved and $\mathcal{Q}$ is quorum subsuming for q.

In our running example Fig. 1, quorum $\{3, 4\}$ is a complete quorum: both of its members are well-behaved and $\mathcal{Q}$ is quorum subsuming for $\{3, 4\}$.

Definition 9 *(Strong Availability)* A quorum system $\mathcal{Q}$ has *strong availability* for a set of well-behaved processes P if every process in P has at least one complete quorum. We call P a *strongly available set* for $\mathcal{Q}$, and and a member of P a *strongly available process*. We say that $\mathcal{Q}$ is strongly available if it is strongly available for a non-empty set.

Intuitively, operations stay available at a strongly available process since its complete quorum can perform operations on her behalf in the face of Byzantine attacks. In Fig. 1, $\mathcal{Q}$ is strongly available for $\{3, 4\}$. In contrast, $\mathcal{Q}$ is only weakly available for process 1: quorum $\{1, 2, 3\}$ includes Byzantine process 2 and quorum $\{1, 4\}$ is not subsumed by $\mathcal{Q}$.

Corollary 2 *Every blocking set of every strongly available process contains at least one strongly available process.*

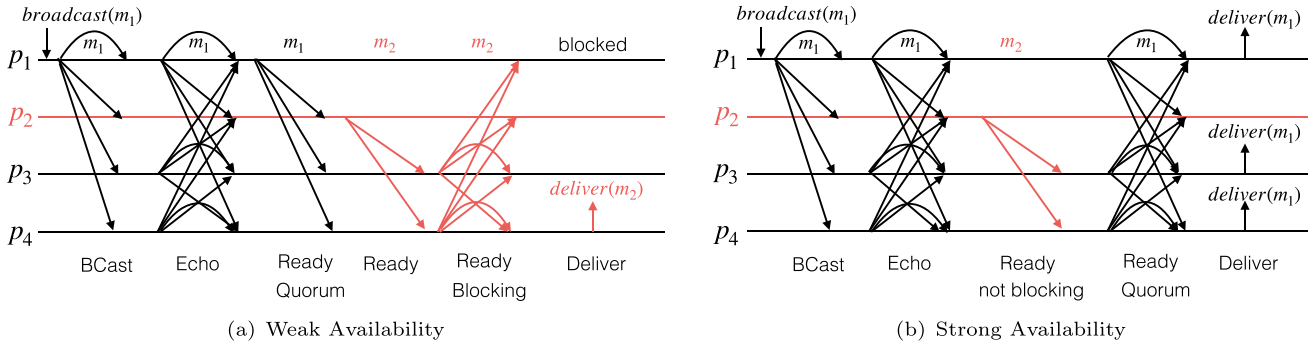

Fig. 2 Quorum Subsumption Supports Progress

This is an immediate corollary of Lemma 1 by taking the strongly available set to be both P and W. Based on this Corollary, protocols for HQSs can benefit from the fact that blocking sets of a strongly available process always include a well-behaved process. In particular, blocking sets are used for amplification in the Byzantine reliable broadcast.

In order to make progress, protocols (such as Bracha's Byzantine reliable broadcast [9]) require the members of a quorum to be able to communicate with at least one of their own quorums (*i.e.*, $2f + 1$ processes), or communicate with a subset of processes that contains at least one well-behaved process (*i.e.*, $f + 1$ processes). Let us see intuitively how quorum subsumption can support liveness properties.

The classic Bracha protocol begins with the sender broadcasting $BCast(m)$. Well-behaved processes broadcast $Echo(m)$ upon receiving it from the sender. After receiving either $2f + 1$ $Echo(m)$ or $f + 1$ $Ready(m)$ messages, processes broadcast $Ready(m)$. Finally, after receiving $2f + 1$ $Ready(m)$ messages, processes deliver m. In Stellar [33] and follow-up works [15, 24, 34], a set of $2f + 1$ processes is generalized to an individual quorum of the process, and considering Corollary 1, a set of $f + 1$ processes is generalized to a blocking set of the process.

As an example, consider a quorum system $\mathcal{Q}$ for processes $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$ where the Byzantine processes are $\{p_2\}$, and $\mathcal{Q}(p_1) = \{\{p_1, p_3, p_4\}\}$, $\mathcal{Q}(p_3) = \{\{p_1, p_2, p_3\}\}$, and $\mathcal{Q}(p_4) = \{\{p_2, p_3, p_4\}\}$. This quorum system has quorum intersection and is weakly available for the set $\{p_1\}$, since there is a well-behaved quorum $\{p_1, p_3, p_4\}$ for the process p_1. We note, however, that $\mathcal{Q}$ is not strongly available for $\{p_1\}$ since p_1 has only the one quorum $q = \{p_1, p_3, p_4\}$, which is not subsumed: p_3 is a member of q, but itself only has quorum $\{p_1, p_2, p_3\}$, which is not a subset of q.

Next, consider the example execution presented in Fig. 2a for intuition on why a quorum system needs stronger conditions than weak availability. Consider that process p_1 issues $broadcast(m_1)$. It sends $BCast(m_1)$ to all processes. Well-behaved processes p_1, p_3, and p_4 send out $Echo(m_1)$ to each other. Process p_1 delivers $Echo(m_1)$ messages from $\{p_1, p_3, p_4\}$ which is a quorum of p_1. Thus, p_1 sends out $Ready(m_1)$ messages. We note that both p_3 and p_4 cannot broadcast $Ready(m_1)$ since they have not received $Echo(m_1)$ from a quorum of their own. Then the Byzantine process p_2 sends $Ready(m_2)$ messages to p_3 and p_4. Since the set $\{p_2\}$ is blocking for the quorums of both processes p_3 and p_4, both send out $Ready(m_2)$ messages. Process p_4 receives $Ready(m_2)$ messages from one of its quorums $\{p_2, p_3, p_4\}$ and delivers m_2. Processes p_1 and p_3 never receive enough $Ready$ messages for either m_1 or m_2 to deliver a message.

Now, consider a slightly different quorum system $\mathcal{Q}$ where $\mathcal{Q}(1) = \{\{p_1, p_3, p_4\}\}$, $\mathcal{Q}(3) = \{\{p_1, p_2, p_3\}, \{p_3, p_4\}\}$, and $\mathcal{Q}(4) = \{\{p_2, p_3, p_4\}, \{p_3, p_4\}\}$. This quorum system is *strongly available* for process p_1: process p_1 has the quorum $\{p_1, p_3, p_4\}$ that is well-behaved, and further, with the addition of the new quorum $\{p_3, p_4\}$, the quorum $\{p_1, p_3, p_4\}$ satisfies the subsumption property. Consider Fig. 2b. After the echo phase, the Byzantine process p_2 still sends the same m_2 message to p_3 and p_4. However, the two processes now have a well-behaved quorum and $\{p_2\}$ is not a blocking set for them. Thus, they do not send a $Ready(m_2)$ message anymore. Further, the members p_3 and p_4 of the well-behaved quorum $q = \{p_1, p_3, p_4\}$ that process p_1 reached out to can now make progress themselves since they have the quorum $\{p_3, p_4\}$ inside q. Processes p_3 and p_4 receive $Echo(m_1)$ messages from their quorum $\{p_3, p_4\}$, and p_1 receives $Echo(m_1)$ messages from its quorum $\{p_1, p_3, p_4\}$. Therefore, they send out $Ready(m_1)$ messages, receive it from each other, and eventually deliver m_1.

3 Protocol implementation

In the subsequent sections, we will see that it is impossible to construct a protocol for Byzantine reliable broadcast and consensus in an HQS given only quorum intersection and quorum availability. After that, we give a protocol for Byzantine reliable broadcast and consensus for an HQS that has quorum intersection and strong availability. We first need

a model of quorum-based protocols, and then the exact specifications of the distributed abstractions we aim to design protocols for. In this section, we consider the former.

We consider a modular design for protocols. A protocol is captured as a component that accepts request events and issues response events. A component uses other components as sub-components: it issues requests to them and accepts responses from them. A component stores a state and defines handlers for incoming requests from the parent component, and incoming responses from children components. Each handler gets the pre-state and the incoming event as input, and outputs the post-state and outgoing events, either as responses to the parent or requests to the children components. The outputs of a handler can be deterministically a function of its inputs, or randomized.

Definition 10 *(Determinism)* A protocol is *deterministic* if the outputs of its handlers are a function of the inputs.

Quorum-based Protocols. A large class of protocols are implemented based on quorum systems. In order to state impossibility results for these protocols, we capture the properties of quorum-based protocols [29, 34] as a few axioms. Our impossibility results concern protocols that adhere to the necessity, sufficiency, and locality axioms.

A process in a quorum-based protocol should process a request only if it can communicate with at least one of its quorums.

Axiom 1 *(Necessity of Quorums [34])* If a well-behaved process p issues a response for a request then there must be a quorum q of p such that p receives at least one message from each member of q.

In a quorum-based protocol, a process only requires the participation of itself and members of one of its quorums to deliver a message.

Axiom 2 *(Sufficiency of Quorums)* For every execution where a well-behaved process p issues a response, there exists an execution where only p and a quorum of p take steps, and p eventually issues the same response.

We add a remark for Byzantine reliable broadcast (BRB) which has a designated sender process. We will use a slight variant of the sufficiency axiom for BRB that states that there exists an execution where only *the sender*, p, and a quorum of p take steps.

The local state of a process is only affected by the information that it receives from the members of its quorums.

Axiom 3 *(Locality [21])* The state of a well-behaved process changes upon receiving a message only if the sender is a member of one of its quorums. Given the same pre-state and incoming messages from quorum members, a well-behaved process produces the same post-state and outgoing messages.

For BRB, we will use a slight variant of the locality axiom that allows processes change state upon receiving messages from *the sender* in addition to members of quorums.

4 Protocol specification

We now define the specification of reliable broadcast and consensus for HQS. The liveness properties are weaker than classical notions since in an HQS, availability might be maintained only for a subset P of well-behaved processes.

Reliable Broadcast. We now define the specification of the reliable broadcast abstraction. The abstraction accepts a single broadcast request from a designated sender (either in the system or a process that is separate from the other processes in the system), and issues delivery responses.

Definition 11 *(Specification of Reliable Broadcast)*

- (Validity for a set of well-behaved processes P). If a well-behaved process broadcasts a message m, then every process in P eventually delivers m.
- (Integrity). If a well-behaved process delivers a message m from a well-behaved sender s, then m was previously broadcast by s.
- (Totality for a set of well-behaved processes P). If a message is delivered by a well-behaved process, then every process in P eventually delivers a message.
- (Consistency). No two well-behaved processes deliver different messages.
- (No duplication). Every well-behaved process delivers at most one message.

We also consider a variant of reliable broadcast that we call *federated reliable broadcast* (FRB). Similar to reliable broadcast, federated reliable broadcast accepts broadcast requests (from processes or externally), and issues delivery responses. In contrast, FRB differs from reliable broadcast in that there can be multiple senders. The specifications for FRB are exactly the same as reliable broadcast, with the exception of validity. The messages that well-behaved processes may not be the same. Therefore, validity only provides guarantees when all senders broadcast the same message, or if there is only one sender.

- *[Federated Reliable Broadcast](Validity for a set of well-behaved processes P). If all senders are well-behaved and broadcast a message m, then eventually every process in P delivers m.*

Consensus. We now consider the specification of the consensus abstraction. It accepts propose requests from processes in the system, and issues decision responses.

Definition 12 *(Specification of Consensus)*

- (Validity). If all processes are well-behaved, and some process decides a value, then that value was proposed by some process.
- (Agreement). No two well-behaved processes decide differently.
- (Termination for a set of well-behaved processes P). Every process in P eventually decides.

5 Impossibility

We now present the impossibility results for consensus and Byzantine Reliable Broadcast (BRB). It is known that quorum intersection and quorum availability are necessary conditions [34] to implement consensus and BRB protocols. In this section, we show that while these two conditions are necessary, they are not sufficient.

We consider information-theoretic settings (Fault axiom [21]) where Byzantine processes have unlimited computational power and can exhibit arbitrary behavior. However, processes communicate only over secure channels so that the recipient knows the identity of the sender, *i.e.*, a Byzantine process cannot impersonate a well-behaved process. This is similar to the classic unauthenticated Byzantine general problem [30], and is advantageous for open decentralized blockchains and HQS, where the trusted authorities including public key infrastructures may not be available.

The two proofs take the same approach. First, we assume towards contradiction there exists a protocol for the abstraction of interest, that satisfies all the desired specifications. We then present a quorum system that satisfies quorum intersection and (weak) availability. After, we consider executions in this quorum system in the face of Byzantine attacks. Through a series of indistinguishable executions, we show the given protocol cannot satisfy all the desired specifications, leading to a contradiction. The high-level idea is that in the information-theoretic setting, a well-behaved process is unable to distinguish between an execution where the sender is Byzantine and sends misleading messages, and an execution where the relaying process is Byzantine and forwards misleading messages. For example, let p_1, p_2 and p_3 be three processes in the system. When p_3 receives conflicting messages from p_1 through p_2, it does not know whether p_1 or p_2 is Byzantine. This eventually leads to violation of the agreement or validity property of the abstraction.

5.1 Consensus

We first consider consensus protocols in HQS. We consider binary proposals. We succinctly represent the values proposed by the processes as a vector of values, which we call a configuration. If the initial value of a process is $\perp$ in the configuration, that process is Byzantine. Otherwise, the process is well-behaved. For example, the configuration $C = \langle 0, 0, \perp \rangle$ denotes the first and second processes proposing zero and the third process being Byzantine.

Theorem 1 *Quorum intersection and weak availability are not sufficient for deterministic quorum-based consensus protocols.*

Proof Towards contradiction, suppose there is a quorum-based consensus protocol that guarantees, for every quorum system $\mathcal{Q}$ with quorum intersection and weak availability, validity, agreement, and termination for P, where P is the set of weakly available processes. Consider a quorum system $\mathcal{Q}$ for processes $\mathcal{P} = \{a, b, c\}$ with the following quorums: $\mathcal{Q}(a) = \{\{a, c\}\}$, $\mathcal{Q}(b) = \{\{a, b\}\}$, $\mathcal{Q}(c) = \{\{b, c\}\}$.

We make the following observations: (1) if all processes are well-behaved, then $\mathcal{Q}$ preserves quorum intersection and weak availability for $\{a, b, c\}$, (2) if only process a is Byzantine, then $\mathcal{Q}$ preserves quorum intersection and weak availability for $\{c\}$, (3) if only process c is Byzantine, then $\mathcal{Q}$ preserves quorum intersection and weak availability for $\{b\}$. Going forward, we implicitly assume termination for weakly available processes.

Now, consider the following four configurations as shown in Fig. 3: $C_0 = \langle 0, 0, 0 \rangle$, $C_1 = \langle 1, 1, 1 \rangle$, $C_2 = \langle 0, 1, \perp \rangle$, and $C_3 = \langle \perp, 1, 1 \rangle$. It suffices to show an execution where at least one property of the protocol is violated. We show a series of executions E_0, E_1 and E_2 over the configurations that lead up to execution E_3 where we arrive at a violation.

- We begin with execution E_0 (shown in red) with the initial configuration C_0. All processes are well-behaved and weakly available by (1). All the messages between a and c are delivered. By termination for weakly available processes and validity, process a decides 0. Additionally, by quorum sufficiency, a reaches this decision with only processes $\{a, c\}$ taking steps.
- Next, we have execution E_1 (shown in blue) with initial configuration C_1. As before, all processes are well-behaved and weakly available by (1). All the messages between b and c are delivered. Again, by termination for weakly available processes and validity, process c decides 1. By quorum sufficiency, c can reach this decision with only processes $\{b, c\}$ taking steps.
- Now, we have execution E_2 as a sequence of E_1 and E_0, with initial configuration C_2. In this case, c is Byzantine and b is weakly available by (3). Suppose messages

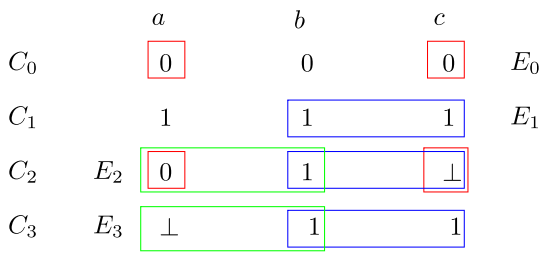

Fig. 3 Indistinguishable Executions

between well-behaved processes a and b are delayed. Byzantine process c first replays E_1 with process b, then replays E_0 with process a. By quorum sufficiency, this causes process a to decide 0. Then, Byzantine process c stays silent. Meanwhile, all messages between a and b are delivered. By termination for b, agreement, and quorum sufficiency, process b decides 0 as well (shown in green).

- Lastly, we have execution E_3 with initial configuration C_3. This time, a is Byzantine and c is weakly available by (2). Suppose messages between b to c are delivered in the beginning. We let processes $\{b, c\}$ replay E_1; thus by locality, c decides 1. Then, Byzantine process a sends messages to b as if it were in the second half of E_2 where the messages between a and b are delivered. In turn, b decides 0. Thus, agreement is violated as two well-behaved processes decided differently, which is a contradiction, and therefore the theorem holds. □

Indistinguishably We provide some intuition for the proof construction. Ultimately, the problem lies in process b not being able to distinguish whether process a or process c is the Byzantine process. More specifically, both E_2 and E_3 begin with execution E_1. Since process b cannot distinguish between the two executions, it does not know which value to decide. If process b believes E_2 is the actual execution, then b should decide 0 to agree with the decision of well-behaved process a. However, if E_3 is the actual execution, then agreement is violated as process c decided 1. Conversely, if process b believes E_3 is the actual execution, then b should decide 1 to agree with the decision of well-behaved process c. Then, if E_2 is the actual execution, agreement is violated as the well-behaved process a decided 0.

We note that this proof could not be constructed if there was quorum subsumption. For example, if the process b adds the quorum $\{a, b, c\}$, then $\mathcal{Q}$ will have quorum subsumption for the quorum $\{a, b, c\}$ of b. However, then by quorum subsumption, there will be no Byzantine processes, and the executions E_2 and E_3 cannot be constructed. If the process a adds the quorum $\{a, b\}$, then it will have quorum subsumption. However, then the process a cannot be a Byzantine process anymore, and the executions E_3 cannot be constructed. Similarly, if the process b adds the quorum $\{b, c\}$, the executions E_2 cannot be constructed.

5.2 Byzantine reliable broadcast

Now, we prove the insufficiency of quorum intersection and quorum availability for Byzantine reliable broadcast.

We consider binary values (from the sender). We represent the initial configuration as an array of values received by the processes from the sender. The sender is a fixed and external process in the executions, and is only used to assign input values for processes in the system, which are captured as the initial configurations. The sender does not take steps in the executions and processes cannot distinguish between executions based on the sender.

Theorem 2 *Quorum intersection and weak availability are not sufficient for deterministic quorum-based reliable broadcast protocols.*

Proof The proof is similar to the proof for consensus. In fact, we reuse the construction in the proof for consensus. There are differences between reliable broadcast and consensus specifications in (1) their validity properties, and (2) their totality and termination properties respectively. The proof is adjusted for these differences. For reliable broadcast, we have a sender s who broadcasts a message. In executions where we want a well-behaved process to deliver the message m, we either (1) keep the sender s well-behaved, have it send m, then apply validity, or (2) have a process deliver m, then apply totality and consistency. The initial configuration represents values received by each process from the sender and implicitly whether the sender is Byzantine (processes receive different values).

Towards contradiction, suppose there is a quorum-based reliable broadcast protocol that guarantees, for every quorum system $\mathcal{Q}$ with quorum intersection and weak availability, integrity, consistency, no duplication, and validity and totality for P, where P is the set of weakly available processes. Executions follow those in the previous proof. Message delivery and delays mirror the previous executions. In execution E_0 for configuration C_0, the well-behaved sender s broadcasts value 0, and messages between processes a and c are delivered. By validity for weakly available processes, process a delivers 0, and by quorum sufficiency, only processes $\{a, c\}$ need to take steps. In execution E_1 for configuration C_1, the well-behaved sender s broadcasts value 1, and messages between processes b and c are delivered. By validity for weakly available processes, process c delivers 1, and quorum sufficiency, only with $\{b, c\}$ taking steps. In configurations C_2 and C_3, the sender s is Byzantine. The messages between processes a and b are delayed in the beginning. In execution E_2 for configuration C_2, the Byzantine sender s and Byzantine process c replay E_1 with process b, then replay E_0 with

process a. By quorum sufficiency, process a delivers 0. Then Byzantine process c stays silent, and messages between processes a and b are delivered. By totality for weakly available processes, since process a delivers a value, process b must also deliver a value. By consistency, process b delivers 0 as well. In the last execution E_3 for configuration C_3, we let the Byzantine process a stay silent in the beginning, and processes b and c replay E_1. Thus by locality, process c delivers 1. Afterwards, messages between process b and c are delayed, and the Byzantine process a replays E_2. As before, process b cannot distinguish between the two executions E_2 and E_3. Since process a sends the exact same messages to process b as the end of E_2, process b delivers 0. Thus, consistency between c and b is violated. □

6 Protocols

We just showed that quorum intersection and availability are not sufficient to implement our desired distributed abstractions. In this section, we show quorum intersection and strong availability, our newly introduced property *are* sufficient to implement both Byzantine reliable broadcast and consensus.

6.1 Reliable broadcast protocol

In this section, we present the Byzantine reliable broadcast (BRB) protocol and a generalization of it called federated reliable broadcast (FRB). We show that quorum intersection and strong availability together are sufficient to implement Byzantine reliable broadcast for HQS. In Alg. 1, we adapt the classical Bracha protocol [9], and highlight the parts that are different from the classical protocol in blue.

Each process stores the set of its individual quorums Q, blocking sets $\mathbb{B}$, and its set of followers $\{$. It also stores the boolean flags *echoed* and *delivered* which record actions the process has taken to avoid duplicate actions. We note here, in contrast to classical Bracha broadcast, that there is no flag for *readied*, *i.e.*, a process can send ready for multiple messages. The reason for this is explained later, after the proof of totality. It uses point-to-point links *ptp* to each of its followers. Upon receiving a request to broadcast a value v (at L. 14), the sender broadcasts the value v to all processes (at L. 16). Upon receiving the message from the sender (at L. 17), a well-behaved process echoes the message among its followers (at L. 20) only if it has not already *echoed*. When a well-behaved process receives a quorum of consistent echo messages (at L. 21), it sends ready messages to all its followers (at L. 24). A well-behaved process also sends a ready message (*i.e.*, amplifies ready) when it receives consistent ready messages from a blocking set (at L. 27). When a well-behaved process receives a quorum of consistent ready messages for v (at L. 29), it delivers v (at L. 31).

The implementation of the federated reliable broadcast (FRB) abstraction is similar. The only difference is that there can be multiple senders (at L. 14): any of the processes can be a sender. A well-behaved sender issues a $broadcast(v)$ request that sends $BCast(v)$ to all processes. In our leader-based consensus protocol, the designated sender in a round is the leader of that round, and eventually a well-behaved leader sends $BCast(v)$ with the same value v to all processes.

Next, we prove the correctness of Alg. 1, assuming the quorum system satisfies the quorum intersection and strong availability properties.

Theorem 3 *Quorum intersection and strong availability are sufficient to implement Byzantine reliable broadcast.*

This theorem follows from Lemmas 4, 6, 5, 7, and 8, in which we show Algorithm 1 guarantees the specifications of Byzantine reliable broadcast. We first prove a couple of helper lemmas.

Lemma 2 *If a process in a complete quorum sends a ready message for m, then a well-behaved process has received a quorum of echo messages for m.*

Proof Consider a process p in a complete quorum q that sends a ready message for m. Since q is a complete quorum, all members of q are well-behaved, *i.e.*, p is well-behaved. A well-behaved process p sends a ready message only if it either (1) receives consistent echo messages from a quorum of p, or (2) receives consistent ready messages from a p-blocking set B. The first case is immediately the conclusion with process p itself. Let us consider the second case. By quorum subsumption, since p is in q, p has a quorum q' that is a subset of q. Since B is a blocking set of p, B intersects with q'. Therefore, B intersects with q as well. Every process in B sent a ready message. Thus, there are processes in q that sent ready messages.

Let p_f be the first process in q that sent a ready message. Applying the same reasoning for p to p_f derives that a blocking set B_f of p_f intersects with q. Thus, p_f has received a ready message from a process in q. This contradicts the definition of p_f above that it is the first process in q to send a ready message. □

Lemma 3 *If a well-behaved process delivers m, then a well-behaved process has received a quorum of echo messages for m.*

Proof Since a well-behaved process p delivered m, it has received a quorum q_p of ready messages. By the strong availability assumption, the set of strongly available process P is non-empty; therefore, there exists a process in P with a complete quorum q. By quorum intersection, q_p and q intersect at a well-behaved process p'. The process p' in the complete quorum q sent a ready message. Thus, by Lemma 2, a we12-

Algorithm 1: Byzantine Reliable Broadcast (BRB) and Federated Reliable Broadcast (FRB)

```
1  Implements: ReliableBroadcast
2    request : broadcast(v)
3    response : deliver(v)
4  Vars:
5    Q : Set[Set[P]]                    ▷ Individual quorums of self
6    𝔹 : Set[Set[P]]                    ▷ blocking sets of self
7    F : Set[P]                         ▷ The followers of self
8    echoed, delivered : Boolean ← false
9    E, R : V ↦ Set[P] ← ∅
10                                      ▷ Set of echoed and readied processes
11 Uses:
12   ptp : PointToPointLink
13     request : send(p, v),  response : deliver(p, v)
14 upon request broadcast(v) from the designated sender
15                                      ▷ (in FRB from any process)
16   ptp request send(p, BCast(v)) for each p ∈ P
17 upon ptp response deliver(p', BCast(v))
18   if ¬echoed then
19     echoed ← true
20     ptp request send(p, Echo(v)) for each p ∈ F
21 upon ptp response deliver(p', Echo(v))
22   E(v) ← E(v) ∪ {p'}
23   if ∃q ∈ Q. q ⊆ E(v) then
24     ptp request send(p, Ready(v)) for
         each p ∈ F
25 upon ptp response deliver(p', Ready(v))
26   R(v) ← R(v) ∪ {p'}
27   if ∃B ∈ 𝔹. B ⊆ R(v) then
28     ptp request send(p, Ready(v)) for each p ∈ F
29   if ¬delivered ∧ ∃q ∈ Q. q ⊆ R(v) then
30     delivered ← true
31     response deliver(v)
```

eceived echo messages for m from at least one of its quorums. □

Lemma 4 *BRB guarantees consistency.*

Proof Suppose two well-behaved processes p_1 and p_2 that deliver two different messages m_1 and m_2, towards contradiction. By Lemma 3, a well-behaved process has received a quorum q_1 of echo messages for m_1, and a well-behaved process has received a quorum q_2 of echo messages for m_2. There is at least a well-behaved process in $q_1 \cap q_2$. However, a well-behaved process sends an echo message only for one message, a contradiction. □

Lemma 5 *BRB guarantees totality for the set of weakly available processes P.*

Proof Consider a well-behaved process p that delivered m. We show that every well-behaved process in $p' \in P$, the set of weakly available processes, eventually delivers m. To do this, we first claim that every well-behaved process p_w sends ready for m to its followers.

Consider any quorum q_w^i of some well-behaved process p_w. Process p delivered m only after receiving a quorum q of ready messages for m. By quorum intersection, q and q_w^i intersect in at least a well-behaved process p_i. Since p_i is in q_w^i, p_i has p_w as a follower. Let I be the union of the processes p_i for all quorums q_w^i of p_w. By construction, the set of processes I is (1) a subset of q, (2) a blocking set of p_w, and (3) its members have p_w as a follower. All well-behaved processes in q send ready for m to their followers. Thus, the set of processes I send ready for m to p_w. This means p_w receives a blocking set of ready for m, and sends ready for m to its followers.

Therefore, every well-behaved process sends ready for m to its followers. By weak availability, every $p' \in P$ has a quorum of only well-behaved processes. Combining these two, p' receives ready for m from every member in one of its quorums, prompting it to deliver m. □

Remark 1 The classical Bracha broadcast and similar protocols include a *readied* boolean variable (at L. 8), for which a process sets to true after sending a ready message (at L. 24), which is used to limit a process to sending at most one ready message by conjuncting the condition $\neg$*readied* to Alg. 23. Our protocol deliberately removes this boolean to extend totality from the set of *strongly* available processes to the set of *weakly* available processes, which we showed with the proof above. For interested readers, we claim (1) the removal of this boolean is necessary to extend totality to the set of weakly available processes, and (2) the protocol including the boolean is still correct though totality is only guaranteed for the set of strongly available processes.

To show (1), we argue that if the *readied* boolean is included, totality is not guaranteed for the set of weakly available processes. Let some well-behaved process deliver a message m. We show some weakly available process p' can send ready for m' where $m' \neq m$. With the boolean, (since p' is in each of its own quorums), p' can never gather a quorum of ready for m and is locked from delivering. For p' to send ready for m', it receives either (a) a quorum of echo for m', or (b) a blocking set of ready for m'. We focus on the latter. Consider some p'-blocking set, B. Although p' is weakly available, meaning B contains at least one well-behaved member p_w, it is not the case that every member of B is weakly available. If p_w is not weakly available, then it can send ready for m' to p' by satisfying (a) or (b) itself. Thus, p' receives ready for m' from every member of B, satisfying (b).

To show (2), all lemmas other than totality remain the same. We argue that if the boolean is included, totality is preserved for the set of strongly available processes. The proof is straightforward; it is sufficient to show no strongly

available process p' can send ready for a message that is not m (as we did in the proof of validity), where m is the message delivered by the well-behaved process. Suppose this were not true, and p' sends ready for m' where $m' \neq m$. By Lemma 2 on p', some well-behaved process p_2 received a quorum q_2 of echo for m'. By Lemma 3 on p, some well-behaved process p_3 received a quorum q_3 of echo for m. However, by intersection, q_2 and q_3 intersect at a well-behaved process. This implies some well-behaved process sent echo for both m and m', a contradiction.

Lemma 6 *BRB guarantees validity for the set of weakly available processes P.*

Proof Consider a well-behaved sender that broadcasts a message m. The set of strongly available processes is assumed to be non-empty; thus let p be a strongly available process with complete quorum q. The strategy is to argue that p will eventually deliver m. Then by Lemma 5, every weakly available process will also deliver m. (This is because in the proof of totality, we actually argue every weakly available process delivers the same message that the initial process delivered. Here, the initial process is p with message m.)

Consider any process $p' \in q$. By quorum subsumption, p' has a quorum $q' \subseteq q$. Since every member of q' is well-behaved, upon receiving m from the sender, every member of q' sends echo m to p'. Thus, p' gathers a quorum of consistent echoes for m, namely from q'. Therefore, p' sends ready for m to its followers, namely p. This directly implies every process in q sent ready m to p, meaning p receiving a quorum of consistent ready messages for m, causing p to deliver m. □

Lemma 7 *BRB guarantees no duplication.*

Proof Since a well-behaved process keeps the *delivered* boolean, it delivers at most one message. □

Lemma 8 *BRB guarantees integrity.*

Proof Consider a well-behaved process p that delivered a message m from a well-behaved sender s. We show that s broadcasted m.

By Lemma 3 on p and m, a well-behaved process has received a quorum q of echo messages for m. By quorum intersection at well-behaved processes, every quorum of a well-behaved process includes a well-behaved process.

Thus, a well-behaved process p_w in q has sent an echo message for m. A well-behaved process sends an echo for m only after receiving it from the sender s. Since the sender s and p_w are well-behaved, and p_w has received the message m from s, by the properties of point-to-point links, s has previously sent m to p_w. Since s is well-behaved, it sends m only upon a broadcast request. □

6.2 Abstract byzantine consensus protocol

In this section, we show that quorum intersection and strong availability are sufficient to implement Byzantine consensus for heterogeneous quorum systems. We first present the consensus protocol, then prove its correctness. The consensus protocol in this section is not amenable to implementation, because it uses infinite state and messages (infinitely many instances of federated reliable broadcast); thus it's called Abstract Byzantine Consensus (ABC). In the next section, we present a more practical protocol that we call Practical Byzantine Consensus (PBC). ABC is simpler than PBC to present the intuitions, and sets the stage for bunching optimizations of PBC.

At a high level, ABC proceeds in rounds with an assigned leader for each. Ballots that carry proposal values are totally ordered. A leader tries to commit its own candidate ballot only after aborting all lower ballots in the system. Leaders use the federated reliable broadcast abstraction (that we saw in §4) to abort or commit ballots. There may be multiple leaders or Byzantine leaders before GST who may broadcast contradicting abort and commit messages for the same ballot. However, by the consistency property of federated reliable broadcast, processes agree on aborting or committing ballots.

A ballot b is a pair $\langle r, v\rangle$ of a round number r and a proposed value v. For a ballot b, the projections $b.r$ and $b.v$ retrieve its round number and value respectively. We define a total order relation $<$ over ballots by first their round numbers, then their values: for two ballots $b = \langle r, v\rangle$ and $b' = \langle r', v'\rangle$, b is *below* b', *i.e.*, $b < b'$ iff $r < r'$, or $r = r' \wedge v < v'$. Additionally, define an equivalence relation $\sim$ on ballots: $b \sim b'$ iff they have the same value, *i.e.*, $v = v'$, in which case we say they are compatible. Otherwise, they are incompatible, which we write as $b \nsim b'$. Finally, we compose these relations: $b \lesssim b'$ iff $b < b' \wedge b \nsim b'$; we say that b is below and incompatible with b'.

For message passing communication, we assume batched network semantics (BNS) [25], where messages issued in an event are sent as a batch, and the receiving process delivers and processes the batch of messages together. (In particular, as we will see later in the correctness proofs, if prepare messages that are sent together are not processed together, then the validity property can be violated.)

ABC is similar to SCP [25, 38] in structure; the important distinction is that ABC uses leaders (based on the leader election module [34]) and guarantees termination. Our protocol guarantees termination regardless of Byzantine processes. On the other hand, the SCP protocol guarantees a liveness property called ∗non-blocking which requires Byzantine processes to stop. (More precisely, if a process p in the *intact* set [24, 38] has not yet decided in some execution, then for

Algorithm 2: Abstract Byzantine Consensus (ABC)

```
1  Implements: Consensus
2    request : propose(v)
3    response : decide(v)
4  Vars:
5    round : ℕ⁺ ← 0                          ▷ Current round number
6    candidate, prepared : ⟨ℕ⁺, V⟩ ← ⟨0, ⊥⟩
7    leader : 𝒫                              ▷ current leader
8  Uses:
9    frb : Ballot ↦ FederatedReliableBroadcast
10   le : EventualLeaderElection
11    response : new-leader(p), request : Complain(round)
12 upon request propose(v)
13   candidate ← ⟨1, v⟩
14   if self = leader then
15     frb(b') request broadcast(𝔸) for all
16     b' ≲ candidate
17 upon frb(b') response deliver (p, 𝔸) for all b' ≲ b where
     prepared < b
18   prepared ← b
19 if self = leader ∧ prepared = candidate then
20   frb(candidate) request broadcast(ℂ)
21 upon frb(b) response deliver(p, ℂ) where
     b = prepared ∧ p = leader
22   response decide(b.v)
23 upon timeout triggered
24   le request Complain(round)
25 upon le response new-leader(p)
26   leader ← p
27   round ← round + 1
28   if self = leader then
29     Delay for time Δ
30   start-timer(round)
31   if prepared = ⟨0, ⊥⟩ then
32     candidate ← ⟨round, candidate.v⟩
33   else
34     candidate ← ⟨round, prepared.v⟩
35   if self = leader then
36     frb(b') request broadcast(𝔸) for all b' ≲ candidate
```

every continuation of that execution in which all the Byzantine processes stop, the process p eventually decides.)

Each process stores four local variables. The variable *round* is the current round number. The variable *candidate* is the ballot that the process tries to commit. The variable *prepared* stores the latest prepared ballot: all ballots lower and incompatible with *prepared* are already aborted. The variable *leader* is the current leader. Each process uses an instance of federated reliable broadcast for each ballot, and accesses an eventual leader election module. The latter accepts *Complain*(*round*) request events and issues *new-leader* response events, and elects a strongly available process as the leader infinitely often. We assume the leader election module chooses a strongly available process infinitely many times. (Previous work [34] presented a probabilistic leader election module, and this protocol can use that module. Future work can investigate the design of more practical protocols for the leader election module with the same interface.)

Upon receiving a *propose* request (at L. 12), a well-behaved process initializes its candidate ballot to the pair of the first round and its own proposal (at L. 13). If the current process self is the *leader*, it tries to prepare its *candidate* by broadcasting abort $\mathbb{A}$ messages for all ballots below and incompatible with *candidate* (at L. 16). When a well-behaved process delivers $\mathbb{A}$ messages from the *leader* for all ballots below and incompatible with some ballot b, and its current *prepared* ballot is below b (at L. 17), it sets *prepared* to b (at L. 18). If the current process self is the *leader*, and the *prepared* ballot is equal to the *candidate* ballot, then it broadcasts a commit $\mathbb{C}$ message for its *candidate* ballot (at L. 20).

When a well-behaved process delivers a $\mathbb{C}$ message for a ballot b from the *leader*, and it has already prepared the same ballot (at L. 21), it issues *decide* for the value of that ballot (at L. 22).

Each round has a duration which is set to a timer at each process at the beginning of the round (at L. 30). The duration of the first round is set to the best guess duration of reaching a decision; however, progress is not dependent on the guess: if a decision is not made in a round, the duration is doubled for the next so that eventually there is enough time for a decision. When a round times out at a process, the process complains to the leader election module (at L. 24). When the leader election module issues a new leader (at L. 25), a well-behaved process updates its *leader* variable, and increments the *round* number (at L. 27). The leader itself then waits for a duration Δ (at L. 29) which we will further explain below. It then updates the *candidate* ballot: if no value is prepared before, the *candidate* ballot is updated to the new round number and the value of the current *candidate* (at L. 32); otherwise, the *candidate* ballot is updated to the new round number and the value of the *prepared* ballot (at L. 34). Then, similar to the steps above, the leader tries to prepare the *candidate* by aborting below and incompatible ballots (at L. 36).

An example execution of Algorithm 2 is described in the appendix [31].

Let us now explain why delay Δ is needed for termination. Without this delay, a Byzantine leader can perform a last minute attack that we illustrate in Fig. 4. Consider that we have four processes $\{l_1, l_2, p_3, p_4\}$, where l_1 is Byzantine, and any set of three processes is a quorum. Let the Byzantine process be the leader l_1, and let the ballot $b = \langle 1, 4\rangle$ be prepared at p_3 and p_4 but not l_2 yet. For b, the leader l_1 sends a a *BCast* message to commit. When p_3 delivers the

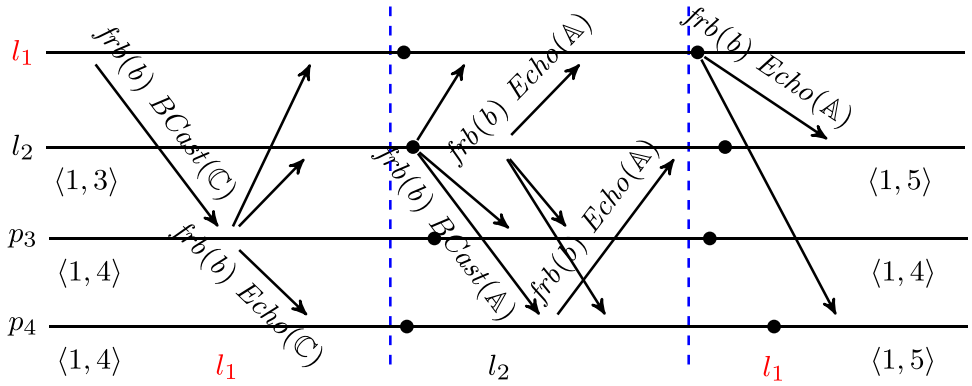

Fig. 4 Last Minute Attack. $b = \langle 1, 4\rangle$. The *candidate* of well-behaved leader l_2 is $b' = \langle 2, 3\rangle$. The votes commit and abort are abbreviated as $\mathbb{C}$ and $\mathbb{A}$. The new leader events are triggered at the black dots at each process. Prepared ballots are shown below the time line for each process

message, since b is prepared at p_3, p_3 echoes commit for it. Then, l_1 doesn't take any other action, and is timed out.

The next well-behaved leader l_2 comes up. Without the delay Δ (at L. 29), the leader l_2 may have not prepared b yet (although other well-behaved processes p_3 and p_4 have). Therefore, the ballot $b' = \langle 2, 3\rangle$ that l_2 updates its *candidate* to (at L. 34) is not be compatible with b. In order to prepare b', the leader l_2 tries to abort b (at L. 36): For b, it broadcasts a *BCast* message to abort. However, b cannot be aborted: in order to abort b, a quorum of processes should echo abort for it. The two well-behaved processes l_2 and p_4 echo abort for it. However, the well-behaved process p_3 has already echoed commit for it, and the Byzantine process l_1 remains silent. Thus, l_2 receives echo messages only from l_2 and p_4 which is not a quorum. Thus, l_2 cannot abort b. Therefore, l_2 eventually times out.

If the next leader is the Byzantine process l_1 again, it can repeat the above scenario: it can abort b and then prepare a higher ballot b_2, and make a well-behaved process echo commit for b_2, before passing the leadership. The attack can continue infinitely, and delay termination.

However, if the leader l_2 waits for the duration Δ (at L. 29), before setting its new *candidate*, it observes the highest prepared ballot b, and adopts its value as the value of its *candidate* b'_2 (at L. 34). Thus, in contrast to b_2, the ballot b'_2 is compatible with b. Therefore, when l_2 tries to commit b'_2, it does not need to abort b. Therefore, it can prepare and commit b'_2, and successfully decide.

The communication delay is bounded after GST. The duration Δ (at L. 29), should be larger than two communication delays. Consider in our example that when l_2 becomes the leader, the well-behaved process p_3 has prepared abort for b. By the totality property of BRB, Lemma 5, the well-behaved process l_2 eventually prepares abort for b as well. As described in the proof of Lemma 5, this takes at most two communication delays so that (1) well-behaved processes receive ready for abort from a blocking set, and broadcast a ready for abort, and (2) l_2 receives a quorum of ready for abort.

We also note that instead of the delay Δ, the above attack can be avoided if the leader election can provide two successive well-behaved leaders.

Next, we prove the correctness of Algorithm 2, assuming the quorum system $\mathcal{Q}$ satisfies the quorum intersection and strong availability properties.

Theorem 4 *Quorum intersection and strong availability are sufficient to implement consensus.*

This theorem follows from the next three lemmas, in which we show Algorithm 2 guarantees the specifications of Byzantine consensus defined in Definition 12.

Lemma 9 *ABC guarantees agreement.*

Proof A well-behaved process decides a value (at L. 22) after delivering commit $\mathbb{C}$ for a ballot with that value (at L. 21). Assume towards contradiction that two well-behaved processes p_1 and p_2 have decided two different values v_1 and v_2 from ballots b_1 and b_2, respectively. Since $v_1 \neq v_2$, we know $b_1 \nsim b_2$. Additionally, as all ballots are totally ordered under $<$, we have either $b_1 \lesssim b_2$ or $b_2 \lesssim b_1$. Without loss of generality, let $b_1 \lesssim b_2$.

Before p_2 decides b_2, it checks that b_2 has been prepared. For p_2 to prepare b_2 (at L. 18), it must find all ballots below and incompatible with b_2 to be aborted (at L. 17). Since $b_1 \lesssim b_2$, p_2 must have delivered abort $\mathbb{A}$ for b_1 before preparing b_2. However, since p_1 decided v_1 with b_1, p_1 must have delivered $\mathbb{C}$ for b_1. Thus, the two processes p_1 and p_2 delivered two different values $\mathbb{A}$ and $\mathbb{C}$ for b_1 respectively, contradicting the consistency property of federated reliable broadcast. □

Lemma 10 *ABC guarantees validity.*

Proof Assuming all processes are well-behaved and that some process has decided a value v, we show that v was proposed by some process. A well-behaved process decides a value (at L. 22) after delivering $\mathbb{C}$ for a ballot with that value (at L. 21). Since all processes are well-behaved (including the leader), by the integrity property of federated reliable broadcast, the leader must have broadcast $\mathbb{C}$ for a ballot (at L. 20),

with value v. A leader broadcasts only her own *candidate* ballot (by condition at L. 19).

We now claim that the value of every *candidate* ballot can be traced back to the proposal of some process. Consider a *candidate* ballot b_c at some process p. The value of b_c is set to either the initial proposal of p (at L. 13 or L. 32), or adopted from the value of its *prepared* ballot (at L. 34). The first case directly affirms our claim: v is the proposal of p (at L. 12).

For the second case (at L. 34), v is assigned the value of p's *prepared* ballot. A well-behaved process only ever updates its *prepared* ballot to some ballot b (at L. 18) after delivering $\mathbb{A}$ for all ballots $b' \lesssim b$ (at L. 17). That is, p must have delivered $\mathbb{A}$ for all such ballots b'. Again by the integrity of federated reliable broadcast, for p to have delivered these $\mathbb{A}$ messages, some leader must have broadcast $\mathbb{A}$ for all such b'. Importantly, leaders broadcast $\mathbb{A}$ for all ballots that are below and incompatible with respect to their own *candidate* ballot (at L. 16 and L. 36).

By the batched network semantics assumption, all messages sent by a process (the leader) in one event to another process (p) are processed and delivered together. Therefore, the ballot b was the *candidate* ballot of a previous leader (at L. 16 or L. 36).

We can extend this reasoning so that the value of ballot b_c is either a proposal, or the value of the *candidate* ballot of yet another previous leader. Since the number of leaders is finite, by well-founded induction, the second case eventually reduces to the first case. □

Lastly, we show our protocol guarantees termination for the set of strongly available processes P. That is, eventually all processes in P decide a value. Intuitively, we combine the totality property of federated reliable broadcast with the guarantee that the leader election module eventually elects a well-behaved process. Then, a well-behaved leader eventually adopts the value of the highest prepared ballot in the system as its candidate, and prepares and commits this candidate.

Before presenting the proof, we consider executions where a new well-behaved leader *cannot abort* a ballot. We saw an example in the first two message rounds in Fig. 4. Let us recall that example. Consider a system of four processes l_1, l_2, p_3, and p_4, one of which (l_1) is Byzantine, and any set of three processes form a quorum. Let the Byzantine process l_1 send a message $m = \mathbb{C}$ to one of the well-behaved processes p_3, and let that process echo it. Thereafter, the Byzantine process l_1 remains silent. Notice that it is impossible for the FRB instance to deliver a message other than m. Even if a new leader l_2 comes, and the remaining two well-behaved processes l_2 and p_4 echo a different message $m' = \mathbb{A}$, they can never form a quorum as one of the well-behaved processes p_3 has already echoed m and well-behaved processes are limited to one echo. If the Byzantine process l_1 breaks its silence and echoes m'. then, a quorum can be formed and the FRB instance can deliver m'.

There are also states where even if Byzantine processes join well-behaved processes, the new leader still cannot make the instance deliver a message different from m. For example, in the same system, let the Byzantine process send m to two well-behaved processes, which they both echo. According to the protocol, a process needs three ready messages to deliver it. Further, in order to send a ready message, a process must receive at least three echoes or two readies. Only the Byzantine process can send a ready. So a process cannot receive two readies. Therefore, to deliver any message $m' \neq m$, at least three processes must echo m'. As two of the four processes have already echoed m, and well-behaved processes only echo once, this is impossible.

Lemma 11 *ABC guarantees termination for the set of strongly available processes.*

Proof When a round ends, the timeout duration for the next round is double that of the previous round. This implies that eventually (after GST), there is sufficient time for well-behaved processes to receive messages and perform necessary actions within a single round. Additionally, the leader election module eventually chooses a well-behaved leader in P, the set of strongly available processes. We show that if any process in P has not already decided a value, then the leader after GST will lead it to decide. We assume from now on that the leader is well-behaved and processes no longer timeout.

Let b_c be the *candidate* ballot of the leader. We first argue that b_c takes on the value of the largest ballot b that has been previously prepared by any well-behaved process (if there is one). At the start of the round, the leader waits Δ time (at L. 29), which is larger than the bound on communication delay (after GST), during which the leader observes (the FRB instance for) every ballot, including b. Since b was prepared by a well-behaved process, then that process delivered $\mathbb{A}$ for all ballots below and incompatible with b. By the consistency and totality properties of FRB, the leader also delivers $\mathbb{A}$ for all ballots below and incompatible with b (at L. 17), and therefore adopts b as its *prepared* ballot (at L. 18). The value of the *candidate* ballot is set to the value in its *prepared* ballot (at L. 34).

The leader tries to abort all ballots that are below and incompatible with b_c (at L. 16 and L. 36). We consider two cases based on whether there is some ballot $b' \lesssim b_c$ that cannot be aborted.

Case 1: *There is no ballot b' whose FRB instance cannot be aborted.* The leader broadcasts abort for all ballots below and incompatible with b_c (at L. 16 and L. 36). After doing so, it broadcasts commit for b_c (at L. 20). By the validity

of federated reliable broadcast, all processes in P eventually deliver commit for b_c (at L. 21), and decide its value (at L. 22).

Case 2: *There is a ballot b' whose FRB instance cannot be aborted.* We first show that if b' cannot be aborted, then b_c and b' are compatible, *i.e.*, have the same value. We already know that b_c and b are compatible. Thus, we just need to show that b and b' are compatible. Since b is the latest prepared ballot, b' is below or equal to b. If b' and b are equal, then they are compatible. Consider the case where b' is below b. For the well-behaved process to have prepared b (at L. 18), it must have delivered $\mathbb{A}$ for all ballots below and incompatible with b (at L. 17). On the other hand, our case assumption is that b' cannot be aborted. Therefore, b is prepared only if b' and b are compatible.

The leader broadcasts to abort ballots below and incompatible with b_c (at L. 16 and L. 36), and then broadcasts commit for b_c (at L. 20). Since b' and b_c are compatible, the leader does not need to abort b'. By the validity of federated reliable broadcast, all processes in P eventually abort all calls below and incompatible with b_c (at L. 36), deliver commit for b_c (at L. 21), and decide its value.

Lastly, for the sake of completeness, if no ballot was previously prepared, then the leader's *prepared* ballot never changes from the initial value (after waiting Δ time post-election). Then, b_c is simply the leader's proposal (at L. 13 and L. 32) and the system follows Case 1. □

7 Practical protocols

In this section, we introduce the Bunched Voting (BV) protocol and the the Practical Byzantine Consensus (PBC) protocol. In contrast to the ABC protocol presented in §6.2, PBC is amenable to implementation. PBC is inspired by the concrete Stellar Consensus protocol in [25]. We remember that ABC uses one instance of Federated Reliable Broadcast (Alg. 1) per ballot. However, there are potentially an infinite number of ballots, implying potentially infinitely many messages and states. Further, since preparing a ballot requires aborting every ballot below and incompatible with that ballot, the set of values that can be proposed must be finite and known beforehand. PBC resolves both of these issues by using a single instance of BV instead. BV batches the echo and ready voting across multiple ballots. More specifically, each process maintains finite state, and only sends and receives a finite number of messages. Lastly, PBC does not rely on the batched network semantics (BNS) assumption [25], which was previously required in ABC.

7.1 Bunched voting

We present the Bunched Voting (BV) protocol in Alg. 3. The BV protocol is comprised of two parts: preparation and commitment. Intuitively, these correspond to how ballots are prepared and committed in the ABC protocol.

Preparation. In ABC, the leader prepares a ballot b by aborting all ballots below and incompatible with b, *i.e.*, broadcasting $\mathbb{A}$ for all FRB instances corresponding to ballots that are below and incompatible with b. Respectively, b is prepared when all ballots below and incompatible have been aborted, *i.e.*, all FRB instances corresponding to ballots that are below and incompatible with b have delivered $\mathbb{A}$. The BV protocol modularizes this process; it accepts the request *prepare*(b) to abort all ballots below and incompatible with b. Respectively, it issues a *prepared*(b) response when all ballots below and incompatible with b have been aborted.

Both the preparation and commitment parts largely resemble FRB in that processes echo, ready, amplify, and finally deliver ballots. To distinguish between the echo and ready messages between the two parts, we use *PEcho* and *PReady* for the former, and *CEcho* and *CReady* for the latter.

Variables. Each process stores the two largest ballots with differing values for which it has sent a *PEcho* message. That is, the *pe* variable tracks the largest ballot it has sent *PEcho* for, while the *pe2* variable tracks the largest ballot it has sent *PEcho* for that is below and incompatible with *pe*. Together, these two variables form what is called the *PEcho summary*, written $\langle pe2, pe\rangle$. This pair summarizes all the ballots that the process has (effectively) sent *abort* for. Similarly, the *PReady* summary, $\langle pr2, pr\rangle$, represents the summary of the ballots that the process has sent *PReady* for.

The process also summarizes the *PEcho* and *PReady* messages that it receives from other processes. The variable *PE* maps each process to the summary of the incoming *PEcho* messages from that process. In other words, let $PE(p) = \langle pe2_p, pe_p\rangle$ at the current process self. The variable pe_p denotes the largest ballot for which self has received a *PEcho* message from p, while $pe2_p$ denotes the largest ballot that is below and incompatible with pe_p for which self has received a *PEcho* message from p. Identically, *PR* maps each process to the summary of incoming *PReady* messages. Lastly, the process stores the largest ballot it has issued a *prepared* response for in the *pd* variable.

Handler for prepare. Upon receiving a *prepare*(b) request (L. 24), the process broadcasts ballot b to all processes (L. 25). Upon receiving this *Prepare*(b) message (L. 26), the process sends a *PEcho*(b) message to its followers (L. 29) if two conditions (L. 27) are satisfied: (a) the round number of b is larger than that of *pe*, and (b) the ballot *ce*, the largest ballot self has previously sent *CEcho* for, is either unset, or it is not below and incompatible with b. If (a) does not hold,

Algorithm 3: Bunched Voting (BV)

```
1  Implements: BunchedVoting
2    request : prepare(b)
3    response : prepared(b)
4    request : commit(b)
5    response : committed(b)
6  Vars:
7    Q                                  ▷ Individual quorums of self
8    F : Set[P]                         ▷ Followers of self
9    PE, PR : P ↦ ⟨Ballot, Ballot⟩ ← ⟨⟨0, ⊥⟩, ⟨0, ⊥⟩⟩
10                                      ▷ {Incoming PEcho and PReady  }
11   pe : Ballot ← ⟨0, ⊥⟩               ▷ Last prepare echoed ballot
12   pe2 : Ballot ← ⟨0, ⊥⟩
13                ▷ 2nd to last incompatible prepare echoed ballot
14   pr : Ballot ← ⟨0, ⊥⟩               ▷ Last prepare readied ballot
15   pr2 : Ballot ← ⟨0, ⊥⟩
16               ▷ 2nd to last incompatible prepare readied ballot
17   pd : Ballot                        ▷ Prepare response ballot
18   CE, CR : P ↦ Ballot ← ⟨0, ⊥⟩
19                              ▷ Incoming CEcho and CReady
20   ce, cd : Ballot ← ⟨0, ⊥⟩
21             ▷ Last commit echoed and commit response ballot
22 Uses:
23   ptp : PointToPointLink
24 upon request prepare(b)
25   ptp request send(Prepare(b)) to ∀p ∈ P
26 upon ptp response deliver(Prepare(b))
27   if pe.r < b.r ∧ (ce = ⟨0, ⊥⟩ ∨ ¬(ce ≲ b)) then
28     ⟨pe2, pe⟩ ← aggregate(⟨pe2, pe⟩, b)
29     ptp request send(self, PEcho(b)) to ∀p ∈ F
30 upon ptp response deliver(p, PEcho(b))
31   PE(p) ← aggregate(PE(p), b)
32   if ∃q ∈ Q. ℓp′ ∈ q. e-advanced(PE(p′), b) then
33     ⟨pr2, pr⟩ ← aggregate(⟨pr2, pr⟩, b)
34     ptp request send(self, PReady(b)) to ∀p ∈ F
35 upon ptp response deliver(p, PReady(b))
36   PR(p) ← aggregate(PR(p), b)
37   if ∃B ∈ blocking(self). ℓp′ ∈ B. e-advanced(PR(p′), b) then
38     ⟨pr2, pr⟩ ← aggregate(⟨pr2, pr⟩, b)
39     ptp request send(self, PReady(b)) to ∀p ∈ F
40   if ∃q ∈ Q. ℓp′ ∈ q. e-advanced(PR(p′), b) then
41     response prepared(b)
42     ptp request send(PRelay(b)) to ∀p ∈ q
43     pd = max(pr, b)
44   if ∃q ∈ Q. ℓp′ ∈ q. e-aborted(PR(p′), ce) then
45     ce ← ⟨0, ⊥⟩
46 upon ptp response deliver(PRelay(b))
47   if e-advanced(⟨pr2, pr⟩, b) then
48     ptp request send(self, PReady(b)) to ∀p ∈ F
49 upon request commit(b)
50   ptp request send(p, Commit(b)) for ßor ∀p ∈ P
51 upon ptp response deliver(p, Commit(b))
52   if b = pd ∧ ¬e-aborted(⟨pe2, pe⟩, b) then
53     ce ← b
54     ptp request send(self, CEcho(b)) to ∀p ∈ F
55 upon ptp response deliver(p, CEcho(b))
56   CE(p) ← max(CE(p), b)
57   if ∃q ∈ Q. ∀p′ ∈ q. CE(p′) = b then
58     ptp request send(self, CReady(b)) to ∀p ∈ F
59 upon ptp response deliver(p, CReady(b))
60   CR(p) ← max(CR(p), b)
61   if ∃B ∈ blocking(self). ∀p′ ∈ B. CR(p′) = b then
62     ptp request send(self, CReady(b)) to ∀p ∈ F
63   if cd = ⊥ ∧ ∃q ∈ Q. ∀p′ ∈ q. CR(p′) = b then
64     cd ← b
65     response committed(b)
66 function aggregate(⟨b2, b⟩, b′)
67   if b < b′ then
68     if b ≁ b′ then
69       b2 ← b
70     b ← b′
71   else if b2 < b′ ∧ b ≁ b′ then
72     b2 = b′
73   return ⟨b2, b⟩
74 function e-aborted(⟨b2, b⟩, b′)
75   return (b′ ≤ b2) ∨ (b′ ≲ b)
76 function e-advanced(⟨b2, b⟩, b′)
77   return (b′ ≤ b2) ∨ (b′.r ≤ b.r ∧ b′ ∼ b)
```

ballot b is stale: self has already sent *PEcho* for a ballot with an equal or larger round number. In particular, this prevents a process from sending multiple *PEcho* messages for the same round. Condition (b) stipulates that if self has attempted to commit a ballot *ce* by sending an echo commit message for it, then self should not simultaneously attempt to prepare a ballot above and incompatible with *ce*. Recall that to prepare a ballot, all ballots below and incompatible with that ballot should be aborted. Therefore, if self sends *PEcho* for a ballot that is above and incompatible with *ce*, it would be trying to abort and commit the *ce* ballot at the same time, which is contradictory. If self indeed sends $PEcho(b)$, then they aggregate b into their own *PEcho* summary (L. 28).

The aggregate subroutine. The *aggregate* subroutine is applied to update all summaries, not just the *PEcho* summary of self. This includes the *PReady* summary of self, as well as the *PEcho* and *PReady* summaries of others (embedded in *PE* and *PR*). The subroutine aggregates a new ballot b' into a summary $\langle b2, b\rangle$. In the resulting summary, b is the largest ballot amongst the three, and $b2$ is the largest remaining ballot that is incompatible with b. The *aggregate* function yields two notable invariants. First, $b2$ is always below and incompatible with b. Second, for every summary, both elements are monotonically non-decreasing.

Effective aborted and advanced. Recall in ABC that for a ballot b to be prepared, all ballots below and incompat-

ible with b must be aborted. This was achieved by having every FRB instance corresponding to a ballot below and incompatible with b deliver $\mathbb{A}$. BV does not maintain separate FRB instances but maintains a similar effect, that we capture as *effectively aborted and advanced*. When a process sends $PEcho(b)$, the process is *effectively* sending echo abort $Echo(\mathbb{A})_{b'}$ for all ballots b' below and incompatible with b. These are not physical messages, hence we say the process has effectively sent them. We say that a process *effectively sent/received echo/ready abort* for b, written as $Echo(\mathbb{A})_{b'}$/$Ready(\mathbb{A})_{b'}$, if it sent/received $PEcho(b)$/$PReady(b)$ for a ballot b such that $b' \lesssim b$. We say that a process p *effectively sent/received echo/ready advance* for b, written as $Echo(\mathbb{A})_{\lesssim b}$/$Ready(\mathbb{A})_{\lesssim b}$, if process p has effectively sent/received $Echo(\mathbb{A})_{b'}$/$Ready(\mathbb{A})_{b'}$ for every ballot b' such that $b' \lesssim b$.

We explained the concepts in the context of echo messages, but are directly lifted to ready messages as well.

Handler for PEcho. When the current process self receives a $PEcho(b)$ message from process p (L. 30), it immediately adds b to the *PEcho* summary of p. Specifically, self sets $PE(p)$ to $aggregate(PE(p), b)$ (L. 31). Then, self checks if it has a quorum q where every member has *effectively sent echo advance* for b. This is done by checking that for every process p' in q, the *e-advanced* predicate holds for the *PEcho* summary of p', and the ballot b. If this condition holds, then self sends $PReady(b)$ (at L. 34), and aggregates b into its own *PReady* summary $\langle pr2, pr\rangle$ (L. 33).

Interestingly, from the *PEcho* summary of a process, we can derive (1) all the ballots that a process effectively echoes advance for, as well as (2) all the ballots that a process has ever effectively sent echo abort for. Let $\langle pe2, pe\rangle$ denote the *PEcho* summary of a process p. For a ballot b, the process p effectively sends $Echo(\mathbb{A})_{\lesssim b}$ if (1*) $(b \leq pe2) \vee (b.r \leq pe.r \wedge b \sim pe)$. Let's see why if this expression holds for b, then p has effectively sent $Echo(\mathbb{A})_{b'}$ for all $b' \lesssim b$. We proceed by disjunction elimination. The case for the right disjunct is trivial; the process p previously sent *PEcho* for pe, effectively sending $Echo(\mathbb{A})_{b*}$ for all $b^* \lesssim pe$. Since b is below or equal, and *compatible* with pe, then it holds that p effectively sent $Echo(\mathbb{A})_{b'}$ for all $b' \lesssim b$. Next, consider the left disjunct. There are two sub-cases, depending on whether b and $pe2$ are compatible. (a) $b \nsim pe2$. Combining (a) with the left disjunct implies $b \lesssim pe2$. Since p previously sent $PEcho(pe2)$, we arrive at the conclusion. (b) $b \sim pe2$. Since $b \leq pe2$ and $pe2 \lesssim pe$, we get $b < pe$. Further, since $b \sim pe2$ and $pe2 \nsim pe$, we get $b \nsim pe$. Combining these, we get $b \lesssim pe$. Since p previously sent $PEcho(pe)$, we arrive at the conclusion.

Notice now that (1*) is precisely the *e-advanced* predicate (at L. 76). That is, given a *PEcho* summary of a process and a ballot, *e-advanced* returns true if the process effectively echo advanced the ballot. In the context of the *PEcho* handler, self uses *e-advanced* to check whether one of its quorums have effectively echo advanced ballot b. If so, it sends $PReady(b)$.

Also from a *PEcho* summary, we can determine (2) all the ballots that a process has ever effectively sent echo abort for. That is, p effectively echoed abort for a ballot b' if (2*) $(b' \leq pe2) \vee (b' \lesssim pe)$. We elide the proof as it is nearly identical the one of (1*). The condition (2*) is precisely the *e-aborted* predicate (at L. 74). Thus, given a *PEcho* summary and a ballot b', *e-aborted* returns true if the process effectively echoed abort for b'. We will use this fact later.

Handler for PReady. When self receives a $PReady(b)$ message from process p (at L. 35), it immediately adds b to the *PReady* summary of p using the *aggregate* function. Specifically, self sets $PR(p)$ to $aggregate(PR(p), b)$ (at L. 36). Afterwards, the *PReady* handler has three parts. In the first part, self checks if they can amplify ready advance for b. Conceptually, this check follows the one described in the *PEcho* handler, with the minor exceptions of using a blocking set instead of a quorum, and using the *PReady* summary of each process rather than the *PEcho* summary of each process. That is, self checks if they have a blocking set B where every member effectively sent ready advance for b, *i.e.*, for every process p in B the *e-advanced* predicate holds for the *PReady* summary of p, and the ballot b. If the condition holds, self amplifies the ready advance by sending $PReady(b)$ (at L. 39), and updates its own *PReady* summary $\langle pr2, pr\rangle$ (L. 38).

For the second part of the *PReady* handler, self checks if it can deliver, *i.e.*, issue a $prepared(b)$ response. Like before, this is done by checking if every member of one of its quorums q effectively sent ready advance for b (L. 40). Formally, self checks if for every process p in q, the *e-advanced* predicate holds for the *PReady* summary of p, and the ballot b. If true, self issues $prepared(b)$ (at L. 41). To draw an analogy to ABC, at this point, every (FRB instance corresponding to a) ballot below and incompatible is aborted. The process retains the largest ballot it has ever issued a *prepared* response for in the *pd* variable (at L. 43). In addition to issuing a response, self sends a $PRelay(b)$ message (at L. 42). We will explain below why the *PRelay* message is critical to maintain the totality property.

The final part of the *PReady* handler is resetting the *ce* variable, which stores the largest ballot self has sent a *CEcho* message for. The *ce* ballot can be reset to $\langle 0, \perp\rangle$ upon learning that an entire quorum q has effectively sent ready abort for *ce*, *i.e.*, self effectively receives $Ready(\mathbb{A})_{ce}$ from every member of one of its quorums. Intuitively, the *ce* ballot represents the ballot that self is trying to commit. However, the quorum q has agreed to effectively abort *ce*, meaning it is impossible to commit that ballot. Therefore, self resets its *ce* variable.

Handler for PRelay. As described in the handler for *PReady* messages, a process p issues a $prepared(b)$ response upon

effectively receiving a quorum q of $Ready(\mathbb{A})_{\precsim b}$ messages. Upon doing so, the process sends a $PRelay(b)$ message to every member of q. When self receives a $PRelay(b)$ message, it checks whether it can effectively sent ready advance for b. If so, then self sends $PReady(b)$. This may seem redundant: since the process p received $Ready(\mathbb{A})_{\precsim b}$ from every member of q, every process in q must have already effectively sent ready advance for b.

The reason for this *PRelay* message and the (re)sending of $PReady(b)$ is to ensure prepare totality for BV — if a well-behaved process issues $prepared(b)$, then every strongly available process is expected to issue $prepared(b)$ as well. (The formal proof of this property is given in Lemma 19.) Recall that when a process sends $PReady(b)$, it effectively sends ready advance for all ballots below and compatible with b. Consider the following scenario: in a quorum q, every process sends $PReady(\langle 4, 5\rangle)$, with the exception of one process, p, that instead sends $PReady(\langle 6, 5\rangle)$. All processes in q effectively send $Ready(\mathbb{A})_{\precsim\langle 4,5\rangle}$, since all processes in q effectively sent ready abort for all ballots below and incompatible with $\langle 4, 5\rangle$. Intuitively, to achieve (prepare) totality, we want p to send *PReady* for precisely the ballot q agreed to effectively advance. That is, $\langle 4, 5\rangle$. Therefore, the *PRelay* message is need to make p send $PReady(\langle 4, 5\rangle)$.

Commitment. After a ballot b has been prepared, it can be committed. Analogizing to ABC again, at this point, all ballots below and incompatible with b have been aborted. The leader then broadcasts $\mathbb{C}$ to the FRB instance corresponding to b. As previously mentioned, the commitment part of BV is similar to FRB; processes echo (*CEcho*), ready (*CReady*), amplify, then finally deliver a ballot.

Variables. Each process stores maps *CE* and *CR* that associate every process p to the largest ballet it has received a *CEcho* and *CReady* message for from p. The *ce* and *cd* variables store the largest ballot for which self has sent a *CEcho* message, and issued a *committed* response for respectively.

Handler for commit. Upon receiving a $commit(b)$ request (L. 49), the process broadcasts a $Commit(b)$ message for ballot b to all processes (at L. 50). Upon receiving the $Commit(b)$ message (at L. 51), the process sends $CEcho(b)$ to its followers (at L. 54) if two conditions (L. 52) are satisfied: (a) b is the largest ballot self has issued a *prepared* response for, and (b) self has not effectively sent echo abort for b. Condition (a) stipulates that self must prepare a ballot before attempting to commit it. Further, since *pd* stores the largest ballot self has prepared, this condition prevents self from attempting to commit stale ballots. Condition (b) mirrors the second condition of to send a *PEcho* message (at L. 27). If $e\text{-}aborted(\langle pe2, pe\rangle, b)$ holds, then self effectively sent $Echo(\mathbb{A})_b$. Therefore, condition (b) requires that self attempts to commit b only if they did not previously attempt to abort b. If both conditions pass, self assigns b to its *ce* variable (at L. 53) and sends $CEcho(b)$ to its followers (at L. 54).

Handlers for CEcho and CReady. Beyond this point, the execution is identical to FRB. When a process receives a quorum of *CEcho* messages for a ballot b, they sends $CReady(b)$ to its followers. Upon receiving a $CReady(b)$ message, self amplifies the ready commit message if they have received a blocking set of $CReady(b)$ messages. Further, if self receives a quorum of *CReady* messages for b and has not yet delivered, it delivers b, *i.e.*, issue a $committed(b)$ response.

Next, we present and prove properties about Bunched Voting that will be useful when proving the correctness of our PBC protocol. Although the properties we prove are closely related to their FRB counterparts, they are defined with critical differences. We first prove several helper lemmas generalizing summaries and their relation to the *e-advanced* and *e-aborted* predicates.

We say a summary $\mathcal{S} = \langle b2, b\rangle$ is *valid* if (p.1) $b2 \precsim b \vee b2 = b = \langle 0, \bot\rangle$. It is straightforward to verify that every summary generated within BV is valid by inspection of the *aggregate* subroutine. Thus the following lemmas apply to all summaries within BV. For Lemmas 12 and 13, let summary $\mathcal{S}_0$ be any valid summary and let $T = (b_1, ..., b_n)$ be any sequence of ballots. Further, let summary $\mathcal{S}_n$ denote the left fold of the *aggregate* function over T starting with $\mathcal{S}_0$.

The following lemma states that if a ballot is aggregated into a summary, the *e-advanced* predicate holds between the resulting summary and that ballot.

Lemma 12 *For all $b_i \in T$, $e\text{-}advanced(\mathcal{S}_n, b_i)$.*

Proof Define summary

$$\mathcal{S}_n :- aggregate(..., aggregate(\mathcal{S}_0, b_1), ..., b_n).$$

We proceed by induction on n, the length of sequence T.

Base case: $n = 0$. Vacuously true as T is empty.

Inductive hypothesis: Let $\mathcal{S}_k = \langle b2, b\rangle$ denote the left fold of *aggregate* over sequence $T_k = (b_1, ..., b_k)$ starting with $\mathcal{S}_0$. Assume $\forall b_i \in T_k$, that $e\text{-}advanced(\mathcal{S}_k, b_i)$. By definition of *e-advanced*, it holds for all $b_i \in T_k$ that

$$(b_i \leq b2) \vee (b_i.r \leq b.r \wedge b_i \sim b). \tag{IH}$$

Inductive step: Let $T_{k+1} = (b_1, ..., b_{k+1})$ and $S_{k+1} = \langle c2, c\rangle :- aggregate(S_k, b_{k+1})$. We must show $\forall b_j \in T_{k+1}.\ e\text{-}advanced(\mathcal{S}_{k+1}, b_j)$. This is equivalent to showing that for all $b_j \in T_{k+1}$,

$$(b_j \leq c2) \vee (b_j.r \leq c.r \wedge b_j \sim c). \tag{Goal}$$

The proof is by case analysis. By inspection of the *aggregate* function, $\mathcal{S}_{k+1}$ can only take on four configurations of ballots, giving us four cases. In each case, b_j is either the newly

aggregated ballot b_{k+1}, or a ballot that was previously aggregated in T_k, giving us two subcases. Further, in each latter subcase, the strategy is to apply disjunctive elimination on (IH), giving us two subsubcases.

Case 1: $\mathcal{S}_{k+1} = \langle c2, c\rangle :- \langle b, b_{k+1}\rangle$. This case occurs when both if-statements on L. 67 and L. 68 are true. More specifically, when $b < b_{k+1}$ and $b \nsim b_{k+1}$. By case (1), we can replace b with $c2$ and b_{k+1} with c.

Subcase a: $b_j := b_{k+1}$. Tautologically, $b_{k+1}.r \leq b_{k+1}.r \wedge b_{k+1} \sim b_{k+1}$. By subcase (a), we replace b_{k+1} with b_j, giving us $b_j.r \leq b_{k+1}.r \wedge b_j \sim b_{k+1}$, By case (1), we replace b_{k+1} with c, giving us $b_j.r \leq c.r \wedge b_j \sim c$, which is the right disjunct of (Goal).

Subcase b: $b_j \in T_k$. Applying disjunctive elimination to (IH), we get two subsubcases. For the following subsubcases, by subcase (b) and (IH), we can replace b_i with b_j.

Subsubcase i: $b_i \leq b2$. By construction, we know $\mathcal{S}_k$ is a valid summary, thereby (p.1), we have $b2 < b$. Transitively with subsubcase (i), we get $b_i \leq b$. By subcase (b), we replace b_i with b_j; by case (1), we replace b with $c2$. This gives us $b_j \leq c2$, which is the left disjunct of (Goal).

Subsubcase ii: $b_i.r \leq b.r \wedge b_i \sim b$. On subsubcase (ii), by subcase (b) we replace b_i with b_j; by case (1), we replace b with $c2$. This gives us $b_j.r \leq c2.r \wedge b_j \sim c2$. This implies $b_j \leq c2$, which is the left disjunct of (Goal).

Case 2: $\mathcal{S}_{k+1} = \langle c2, c\rangle :- \langle b2, b_{k+1}\rangle$. This case occurs when the if-statement on L. 67 is true but the if-statement on L. 68 is false. More specifically, when (2.1) $b < b_{k+1} \wedge b \sim b_{k+1}$. By case (2), we can replace $b2$ with $c2$ and b_{k+1} with c.

Subcase c: $b_j :- b_{k+1}$. This is exactly the same as case (1) subcase (a).

Subcase d: $b_j \in S_k$. Applying disjunctive elimination to (IH), we get two subsubcases. For the following subsubcases, by subcase (d) and (IH), we can replace b_i with b_j.

Subsubcase iii: $b_i \leq b2$. Replacing on subsubcase (iii) with subcase (d) and case (2), we get $b_j \leq c2$, which is the left disjunct of (Goal).

Subsubcase iv: $b_i.r \leq b.r \wedge b_i \sim b$. By the left conjuncts of subsubcase (iv) and (2.1), we have $b_i.r \leq b.r \leq b_{k+1}.r$. By the right conjuncts of subsubcase (iv) and (2.1), we have $b_i \sim b \sim b_{k+1}$. Combining these together, we get $b_i.r \leq b_{k+1}.r \wedge b_i \sim b_{k+1}$. Replacing with subcase (d) and case (2), we get $b_j.r \leq c \wedge b_j \sim c$, which is the right disjunct of (Goal).

Case 3: $\mathcal{S}_{k+1} = \langle c2, c\rangle :- \langle b_{k+1}, b\rangle$. This case occurs when the if-statement on L. 67 is false and the else-if-statement on L. 71 is true. More specifically, when (3.1) $b2 < b_{k+1} \wedge b \nsim b_{k+1}$. By case (3), we can replace b_{k+1} with $c2$ and b with c.

Subcase e: $b_j := b_{k+1}$. Tautologically, $b_j \leq b_j$. By subcase (e) and case (3), we replace b_j with $c2$, giving us $b_j \leq c2$, which is the left disjunct of (Goal).

Subcase f: $b_j \in S_k$. Applying disjunctive elimination to (IH), we get two subsubcases. For the following subsubcases, by subcase (f) and (IH), we can replace b_i with b_j.

Subsubcase v: $b_i \leq b2$. By subsubcase (v) and (3.1), we get $b_i \leq b2 < b_{k+1}$. By subcase (f), we replace b_i with b_j; by case (3), we replace b_{k+1} with $c2$, giving us $b_j \leq c2$, which is the left disjunct of (Goal).

Subsubcase vi: $b_i.r \leq b.r \wedge b_i \sim b$. On subsubcase (vi), by subcase (f), we replace b_i with b_j; by case (3), we replace b with c, giving us $b_j.r \leq c.r \wedge b_j \sim c$, which is the right disjunct of (Goal).

Case 4: $\mathcal{S}_{k+1} = \langle c2, c\rangle :- \langle b2, b\rangle$. The last case is when none of the if-statements are true, *i.e.*, no changes are made. By case (4), we can replace $b2$ with $c2$ and b with c.

Subcase g: $b_j := b_{k+1}$. There are two subsubcases, depending on if $b_{k+1} \sim b$. By subcase (g), we can replace b_{k+1} with b_j.

Subsubcase vii: $b_{k+1} \sim b$. Since the if-statement on L. 67 is false, we know $b_{k+1} \leq b$, which implies $b_{k+1}.r \leq b.r$. Combining this with subsubcase (vii), we get $b_{k+1}.r \leq b.r \wedge b_{k+1} \sim b$. By subcase (g), we replace b_{k+1} with b_j; by case (4), we replace b with c, giving us $b_j.r \leq c.r \wedge b_j \sim c$, which is the right disjunct of (Goal).

Subsubcase viii: $b_{k+1} \nsim b$. Since the if-statement on L. 71 is false, by subsubcase (viii), we get $b_{k+1} \leq b2$. By subcase (g), we replace b_{k+1} with b_j; by case (4), we replace $b2$ with $c2$, giving us $b_j \leq c2$, which is the left disjunct of (Goal).

Subcase h: $b_j \in S_k$. Trivial by case (4) and (IH).

As these cases and their (sub)subcases are exhaustive over the behavior of *aggregate* and all $b_j \in T_{k+1}$, the inductive step is complete. □

The following lemma states that if the *e-advanced* predicate holds between a summary and a ballot b, when additional ballots are aggregated into the summary, *e-advanced* continues to hold between the latest summary and b.

Lemma 13 *For any ballot* b^*, *e-advanced*$(\mathcal{S}_0, b^*) \Rightarrow$ *e-advanced*$(\mathcal{S}_n, b^*)$.

Proof Define summary

$$\mathcal{S}_n :- \mathit{aggregate}(..., \mathit{aggregate}(\mathcal{S}_0, b_1), ..., b_n).$$

Fix an arbitrary b^* such that *e-advanced*$(\mathcal{S}_0, b^*)$ holds. We proceed by induction on n, the length of sequence T.

Base case: $n = 0$. Trivially holds as $\mathcal{S}_n = \mathcal{S}_0$, for which we know *e-advanced*$(\mathcal{S}_0, b^*)$ holds.

Inductive hypothesis: Let $\mathcal{S}_k = \langle b2, b\rangle$ denote the left fold of *aggregate* over sequence $T_k = (b_1, ..., b_k)$ starting with $\mathcal{S}_0$. Assume *e-advanced*$(\mathcal{S}_k, b^*)$. By definition of *e-advanced*,

this is logically equivalent to

$$(b^* \leq b2) \vee (b^*.r \leq b.r \wedge b^* \sim b). \tag{IH}$$

Inductive step: Let $\mathcal{S}_{k+1} = \langle c2, c\rangle :- aggregate(\mathcal{S}_k, b_{k+1})$, for an arbitrary ballot b_{k+1}. We must show *e-advanced* $(\mathcal{S}_{k+1}, b^*)$ holds, which is logically equivalent to

$$(b^* \leq c2) \vee (b^*.r \leq c.r \wedge b^* \sim c). \tag{Goal}$$

We apply disjunctive elimination on (IH) towards proving (Goal), giving us two cases.

Case 1: $b^* \leq b2$. By inspection of *aggregate*, we see that $c2$ has three possible values, giving us three subcases.

Subcase a: $c2 :- b$. This subcase occurs when both if-statements on L. 67 and L. 68 are true. By construction, we know $\mathcal{S}_k$ is a valid summary, thereby (p.1), we have $b2 < b$. By case (1), transitively, we get $b^* \leq b$. By subcase (a), we replace b with $c2$, giving us $b^* \leq c2$, which is the left disjunct of (Goal).

Subcase b: $c2 :- b_{k+1}$. This subcase occurs when the else-if-statement on L. 71 is true. By the left conjunct of the condition, we know $b2 < b_{k+1}$. Combining this with case (1), we get $b^* \leq b_{k+1}$. By subcase (b), we replace b_{k+1} with $c2$, giving us $b^* \leq c2$ which is the left disjunct of (Goal).

Subcase c: $c2 :- b2$. This subcase occurs in all other scenarios, *i.e.*, no changes are made. By subcase (c), we replace $b2$ with $c2$ on case (1), giving us $b^* \leq c2$, which is the left disjunct of (Goal).

Case 2: $b^*.r \leq b.r \wedge b^* \sim b$. By inspection of *aggregate*, we see that c has two possible values, giving us two subcases.

Subcase d: $c :- b_{k+1}$. This subcase occurs when the if-statement on L. 67 is true. That is, (d.1) $b < b_{k+1}$. There are two further subsubcases, depending on if $b \sim b_{k+1}$.

Subsubcase i: $b \sim b_{k+1}$. By case (2), we know $b^*.r \leq b.r$; transitively with (d.1), we have (i.1) $b^*.r \leq b.r \leq b_{k+1}.r$. Again by case (2), we know $b^* \sim b$; by subsubcase (i), we have (i.2) $b^* \sim b \sim b_{k+1}$. Combining (i.1) and (i.2), we have $b^*.r \leq b_{k+1}.r \wedge b^* \sim b_{k+1}$. By subcase (d), we replace b_{k+1} with c, giving us $b^* \leq c.r \wedge b^* \sim c$, which is the right disjunct of (Goal).

Subsubcase ii: $b \nsim b_{k+1}$. By (d.1) and subsubcase (ii), we know the if-statement on L. 68 is also true. That is, (ii.1) $c2 = b$. Case (2) directly implies $b^* \leq b$. By (ii.1), we replace b with $c2$, giving us $b^* \leq c2$, which is the left disjunct of (Goal).

Subcase e: $c :- b$. This subcase occurs when the if-statement on L. 67 is false, *i.e.*, no changes are made. Replacing b with c in case (2), we get $b^*.r \leq c.r \wedge b^* \sim c$, which is the left disjunct of (Goal).

Since *e-advanced* $(\mathcal{S}_{k+1}, b^*)$ holds in all (sub)cases, the inductive step is complete. □

Lemma 14 *For any summary $\mathcal{S}$ and ballot b, if e-advanced $(\mathcal{S}, b)$, then $\forall b^*.\ b^* \lesssim b \Rightarrow$ e-aborted $(\mathcal{S}, b^*)$.*

Proof Let $\mathcal{S} = \langle b2, b\rangle$ be any summary. Fix two arbitrary ballots b_a and b^* such that *e-advanced* $(\mathcal{S}, b_a)$ and

$$b^* \lesssim b_a. \tag{1}$$

We must show that *e-aborted* $(\mathcal{S}, b^*)$ holds, which is equivalent to

$$(b^* \leq b2) \vee (b^* \lesssim b). \tag{Goal}$$

Since *e-advanced* $(\mathcal{S}, b_a)$ holds, we know

$$(b_a \leq b2) \vee (b_a.r \leq b.r \wedge b_a \sim b). \tag{2}$$

We apply disjunctive elimination on Eq. 2 towards proving (Goal), giving us two cases.

Case 1: $b_a \leq b2$. By Eq. 1, we know $b^* < b_a$. Combining this with case (1), transitively, we get $b^* \leq b2$, which is the left disjunct of (Goal).

Case 2: $b_a.r \leq b.r \wedge b_a \sim b$. By Eq. 1, we know $b^* < b_a$. Combining this with case (2), transitively, we get (2.1) $b^*.r \leq b.r$. Again by Eq. 1, we know $b^* \nsim b_a$. Combining this with case (2), we get (2.2) $b^* \nsim b$. Together, (2.1) and (2.2) imply that $b^* \lesssim b$, which is the right disjunct of (Goal). □

Lemma 15 *If a strongly available process sends CReady(b), then no well-behaved process issues prepared(b') such that b is below and incompatible with b', i.e., $b \lesssim b'$.*

Proof Let p be a process in P, the set of strongly available processes. Further, let p send *CReady*(b). For p to send *CReady*(b), it must gather either a (1) a quorum of *CEcho*(b) messages, or (2) a blocking set of *CReady*(b) messages. By Corollary 2, (2) reduces to (1) in that there must be a first process in P that received a quorum q of *CEcho*(b) messages. For simplicity, we maintain this process to be p.

Now suppose a well-behaved process p' issues a *prepared*(b') response such that $b \lesssim b'$, towards contradiction. For p' to issue *prepared*(b'), it effectively received $Ready(\mathbb{A})_{\lesssim b'}$ from a quorum q' (L. 40). We know P is non-empty; therefore, by intersection, q' contains a strongly available process. For this strongly available process to effectively send $Ready(\mathbb{A})_{\lesssim b'}$, it must effectively receive either a quorum of $Echo(\mathbb{A})_{\lesssim b'}$ (L. 32), or a blocking set of $Ready(\mathbb{A})_{\lesssim b'}$ (L. 37). Reapplying the reasoning between (2) and (1) here, there is a first process p_0 in P that effectively received $Echo(\mathbb{A})_{\lesssim b'}$ from a quorum q_0. By quorum

intersection, there is a well-behaved process p_w between q and q_0. That is, p_w effectively sent $Echo(\mathbb{A})_{\lesssim b'}$ as well as $CEcho(b)$. There are two cases, based on which message p_w sent first.

Case 1: p_w *effectively sends* $Echo(\mathbb{A})_{\lesssim b'}$ *before* $CEcho(b)$. Let $\langle pe2_0, pe_0\rangle$ denote the *PEcho* summary of p_w after effectively sent $Echo(\mathbb{A})_{\lesssim b'}$. Let $\langle pe2, pe\rangle$ denote the *PEcho* summary of p_w before sending $CEcho(b)$. A well-behaved process aggregates every *PEcho* message they send (at L. 28). Then, since *e-advanced*$(\langle pe2_0, pe_0\rangle, b')$ holds, by Lemma 13, *e-advanced*$(\langle pe2, pe\rangle, b')$ holds as well. Combining this with the fact that $b \lesssim b'$, by Lemma 14, *e-aborted*$(\langle pe2, pe\rangle, b)$ holds. Now, consider the condition that must hold for p_w to send $CEcho(b)$ (at L. 52); specifically, $\neg$*e-aborted*$(\langle pe2, pe\rangle, b)$ must be true. Thus, we have a contradiction.

Case 2: p_w *sends* $CEcho(b)$ *before effectively sending* $Echo(\mathbb{A})_{\lesssim b'}$. When p_w sends $CEcho(b)$, the *ce* variable of p_w is set to b (L. 53). We begin by stating two claims that will be used to prove this case, but defer their proofs until the end.

Claim 1 For p_w to effectively send $Echo(\mathbb{A})_{\lesssim b'}$, it first reset its *ce* ballot to $\langle 0, \perp\rangle$.

Claim 2 For any valid summary $\mathcal{S}$ and ballot d, if *e-aborted* $(\mathcal{S}, d)$, then $\exists b^*.\ (d \lesssim b^*) \wedge$ *e-advanced*$(\mathcal{S}, b^*)$.

By claim (1), before p_w effectively sends $Echo(\mathbb{A})_{\lesssim b'}$, it first reset its *ce* ballot to $\langle 0, \perp\rangle$. Then without loss of generality, let p_w be the first process in q that had reset its *ce*. By the condition at L. 44, p_w resets *ce* when it has effectively received $Ready(\mathbb{A})_b$ from a quorum. In the same way as before, this can be tracked back to a process in P that effectively received $Echo(\mathbb{A})_b$ from a quorum q_1. By quorum intersection, there is a well-behaved process p_v between q and q_1. Thus, p_v sent both $CEcho(b)$ as well as $Echo(\mathbb{A})_b$. By claim (2), since p_v sent $Echo(\mathbb{A})_b$, it effectively sent $Echo(\mathbb{A})_{\lesssim b^*}$ for some ballot b^* such that $b \lesssim b^*$. Thus, the same two cases as above are repeated for p_v; however, in the second case, since p_w was the first process in q to reset its *ce* variable, p_v could not have reset its *ce* variable, resulting in a contradiction.

It remains to prove claims (1) and (2). Consider claim (1) first. As mentioned, when p_w sends $CEcho(b)$, the *ce* variable of p_w is set to b (L. 53). Consider the condition to send a *PEcho* message at L. 27, specifically the right conjunct. We argue that if p_w effectively sends $Echo(\mathbb{A})_{\lesssim b'}$, it must send $PEcho(b^*)$, for some b^* such that $b \lesssim b^*$. However, since $ce = b$, the rightmost disjunct of the aforementioned condition prevents p_w from sending such a b^*. Therefore, the right conjunct can only be satisfied by resetting *ce* to $\langle 0, \perp\rangle$.

Suppose p_w effectively sends $Echo(\mathbb{A})_{\lesssim b'}$. Let $\mathcal{S} = \langle pe2, pe\rangle$ be the *PEcho* summary of p_w after doing so. Then we know *e-advanced*$(\mathcal{S}, b')$ holds. This is equivalent to $(b' \leq pe2) \vee (b'.r \leq pe.r \wedge b' \sim pe)$. We apply disjunctive elimination, towards proving p_w sent *PEcho* for some b^* such that $b \lesssim b^*$.

Case A: $b' \leq pe2$. Combining case (A) with the fact that $b \lesssim b'$, transitively, we get $b < pe2$. We have two subcases, depending on whether $b \sim pe2$.

Subcase i: $b \nsim pe2$. Combining case (A) with subcase (i), we get $b \lesssim pe2$. We know p_w previously sent $PEcho(pe2)$.

Subcase ii: $b \sim pe2$. Since $\mathcal{S}$ is a valid summary, by (p.1), we get $pe2 \lesssim pe$. Combining this with (A), we get $b < pe2 < pe$. Further, by subcase (ii) we get $b \sim pe2$, but $pe2 \nsim pe$; therefore, $b \nsim pe$. Together, this implies $b \lesssim pe$. And we know p_w previously sent $PEcho(pe)$.

Case B: $b'.r \leq pe.r \wedge b' \sim pe$. We know $b \lesssim b'$. Combining this with case (B), we have $b < b' \leq pe$. Since $b \nsim b'$ and $b' \sim pe$, we get $b \nsim pe$. Together, these imply $b \lesssim pe$. And we know p_w previously sent $PEcho(pe)$.

Next, consider claim (2). The proof is by a series of observations. For any valid summary $\langle c2, c\rangle$ and ballot d,

I. *e-aborted*$(\langle c2, c\rangle, d) \Rightarrow (d \lesssim c2 \vee d \lesssim c)$. Straightforward by case analysis.
II. *e-advanced*$(\langle c2, c\rangle, c2)$, *e-advanced*$(\langle c2, c\rangle, c)$. Follows by Lemma 12.
III. if *e-aborted*$(\langle c2, c\rangle, d)$ then $\exists b^*.\ (d \lesssim b^*) \wedge$ *e-advanced* $(\mathcal{S}, b^*)$. Follows from (I) and (II).

Thereby proving claims (1) and (2), the proof is complete. □

Lemma 16 (BV Commit Consistency) *No pair of well-behaved processes issue committed responses for incompatible ballots.*

Proof A well-behaved process issues a *committed* response for a ballot after receiving a quorum of *CReady* messages for that ballot. Let processes p_1 and p_2 both issue a *committed* response for ballots b_1 and b_2 with quorums q_1 and q_2, respectively. Consider only p_1 and q_1, for now. We know the set of strongly available processes P is non-empty; therefore, by intersection, q_1 contains a strongly available process p'. Accordingly, for p' to send $CReady(b_1)$ to p_1, p' receives either (1) a quorum of $CEcho(b_1)$, or (2) a blocking set of $CReady(b_1)$. By Corollary 2, (2) reduces to (1) in that there must be a first process in P that received a quorum of $CEcho(b_1)$.

We then apply the same reasoning to p_2 and q_2. Thus, we have that a well-behaved process received a quorum of $CEcho(b_1)$ and that a (possibly different) well-behaved process received a quorum of $CEcho(b_2)$. By intersection, these two quorums intersect at a well-behaved process p_w, which sent *CEcho* for both b_1 and b_2. A well-behaved process only sends *CEcho* for a ballot if that ballot is currently prepared,

by the condition at L. 52. Therefore, p_w must have issued a *prepared* response for both p_1 and p_2.

In so far, we have: (a) p_1 and p_2 both received *CReady* from a process in P, and (b) some well-behaved process p_w issued a *prepared* response for both b_1 and b_2. Towards contradiction, assume b_1 and b_2 are incompatible. Without loss of generality, let $b_1 < b_2$. Since $b_1 \nsim b_2$, we have (c) $b_1 \lesssim b_2$. From (a), a process in P sent $CReady(b_1)$, and from (b), a well-behaved process issued $prepared(b_2)$. By Lemma 15 and (c), we reach a contradiction. □

Lemma 17 (BV Prepare Integrity) *If all processes are well-behaved, and a process issues a* $prepared(b)$ *response, then some process requested* $prepare(b)$.

Proof Let well-behaved process p issue $prepared(b)$ for ballot b. Since all processes are well-behaved, we can simply trace b back to a *prepare* request as follows. For p to issue a $prepared(b)$, it effectively received a quorum q_3 of $Ready(\mathbb{A})_{\lesssim b}$ (L. 40). Importantly, at least one member of $p_3 \in q_3$ must have sent precisely $PReady(b)$ (L. 35). For this to happen, p_3 must have effectively received either (1) a quorum of $Echo(\mathbb{A})_{\lesssim b}$ (L. 32), (2) a blocking set of $Ready(\mathbb{A})_{\lesssim b}$ (L. 37), or (3) a $PRelay(b)$ message.

For case (3), a $PRelay(b)$ message is sent only after issuing a $prepared(b)$ response (L. 42). Thus, (3) reduces to (1) and (2). Next, consider (2); if p_3 effectively received a blocking set of $Ready(\mathbb{A})_{\lesssim b}$, there is at least one blocking set member p_2 that sent precisely $PReady(b)$ (L. 35). Then, either (1) or (2) must hold for p_2 as well. Repeating this reasoning, since every process is well-behaved, (2) eventually reduces to (1) in that there must be a first process that effectively gathered a quorum of $Echo(\mathbb{A})_{\lesssim b}$. For simplicity, we maintain this process to be p_2, *i.e.*, p_2 has a quorum q_1 for which every member p_1 of q_1, $e\text{-}advanced(PE(p_1), b)$ holds (L. 32). At least one of these members must have sent precisely $PEcho(b)$ (L. 30). Lastly, a process sends $PEcho(b)$ only if some process requested $prepare(b)$. □

Lemma 18 (BV Commit Integrity) *If all processes are well-behaved, and a process issues a* $committed(b)$ *response, then some process requested* $commit(b)$.

Proof Let a well-behaved process p issue $committed(b)$ for a ballot b. Since all processes are well-behaved, we can simply trace b back to a *commit* request as follows. For p to issue $committed(b)$, it must have received a quorum q of $CReady(b)$ messages (L. 63). Then, for every quorum member p_3 of q, since p_3 sent $CReady(b)$, it must have received either a (1) quorum of $CEcho(b)$ (L. 57) or (2) a blocking set of $CReady(b)$ (L. 61).

Using the same argument as the proof of Lemma 17, (2) eventually reduces to (1) in that there must be a first process that gathered a quorum of $CEcho(b)$. Lastly, a well-behaved process only sends $CEcho(b)$ if a process requested $commit(b)$. □

Lemma 19 (BV Prepare Totality) *If a well-behaved process issues* $prepared(b)$*, then every strongly available process eventually issues* $prepared(b)$.

Proof Consider a well-behaved process p_0 that has issued a $prepared(b)$ response after effectively receiving a quorum q_0 of $Ready(\mathbb{A})_{\lesssim b}$. We show that every process $p \in P$, where P is the set of strongly available processes, eventually issues $prepared(b)$ as well. Consider the complete quorum of p, say q. The proof is in three parts. We first claim that (1) every member p' of q eventually receives a blocking set I of $PReady(b)$ messages.

Consider any $p' \in q$ and any one of its quorums, say q'. By quorum intersection, q_0 and q' intersect at some well-behaved process p_i. Let I be the union of these processes p_i over every quorum of p'. That is, I is the set of well-behaved processes that are in both q_0 and a quorum of p'. By construction, we have $I \subseteq q_0$ and I is a blocking set of p'.

After p_0 issues $prepared(b)$, it sends $PRelay(b)$ to all processes in q_0 (L. 42). Therefore, every process in I received $PRelay(b)$. Then to prove claim (1), by the condition on L. 47, it is sufficient to show that $e\text{-}advanced(\langle pr2, pr\rangle, b)$ holds, for every member of I. Consider any process $p_i \in I$. Let $\langle pr2_i, pr_i\rangle$ denote the *PReady* summary of p_i when it receives the $PRelay(b)$ message. Since p_0 issued $prepared(b)$, we know p_i had effectively sent $Ready(\mathbb{A})_{\lesssim b}$. If p_i sent any *PReady* messages after those, all of which are aggregated (L. 33 and L. 38), then by Lemma 13, the *PReady* summary continues to effectively advance b. Therefore, $\langle pr2_i, pr_i\rangle$ holds. Thus, the blocking set I of p' send $PReady(b)$ messages.

Next, we claim (2) every member p' of q sends $PReady(b)$ to p. By claim (1), consider when p' receives a $PReady(b)$ message from a member p_i of the blocking set I. At L. 36, p' aggregates $PReady(b)$ into $PR(p_i)$. Then by Lemma 12, we know $e\text{-}advanced(PR(p_i), b)$ holds. In other words, p' effectively receives $Ready(\mathbb{A})_{\lesssim b}$ from p_i. Therefore, upon receiving $PReady(b)$ from the last member of I, the process p' receives a blocking set of $Ready(\mathbb{A})_{\lesssim b}$, satisfying the condition at L. 37. Thus, p' sends $PReady(b)$ to p (L. 39), proving claim (2).

For the final step of the proof, we argue that p indeed issues $prepared(b)$ upon receiving these $PReady(b)$ messages from every member p' of q. The argument is similar to that of claim (2). Upon receiving $PReady(b)$ from each p', p aggregates each message into $PR(p')$. By Lemma 12, we know $e\text{-}advanced(PR(p'), b)$ holds. Upon receiving $PReady(b)$ from the last member p' of q, p effectively gathers a quorum of $Ready(\mathbb{A})_{\lesssim b}$, satisfying the condition at L. 40, causing p to issue $prepared(b)$. □

Algorithm 4: Practical Byzantine Consensus (PBC)

```
1  Implements: Consensus
2    request : propose(v)
3    response : decide(v)
4  Vars:
5    round : ℕ⁺ ← 0                          ▷ Current round number
6    candidate, prepared : ⟨ℕ⁺, V⟩ ← ⟨0, ⊥⟩
7    leader : 𝒫                              ▷ current leader
8  Uses:
9    bv : BunchedVoting
10   le : EventualLeaderElection
11   ptp : PointToPointLink
12 upon request propose(v)
13   candidate ← ⟨1, v⟩
14   if self = leader then
15     bv request prepare(candidate)
16 upon bv response prepared(b) where prepared < b
17   prepared ← b
18   if self = leader ∧ prepared = candidate then
19     bv request commit(candidate)
20 upon bv response committed(b) from p where
     b = prepared ∧ p = leader
21   response decide(b.v)
22 upon timeout triggered
23   le request Complain(round)
24 upon le response new-leader(p)
25   leader ← p
26   round ← round + 1
27   if self = leader then
28     Delay for time Δ
29   start-timer(round)
30   if prepared = ⟨0, ⊥⟩ then
31     candidate ← ⟨round, candidate.v⟩
32   else
33     candidate ← ⟨round, prepared.v⟩
34   bv request new-round(round)
35   if self = leader then
36     bv request prepare(candidate)
```

7.2 Practical byzantine consensus protocol

In this section, we present a Practical Byzantine Consensus protocol (PBC) called Satrapy using the Bunched Voting (BV) protocol (Alg. 4). The protocol is identical to the ABC protocol presented above (Alg. 2), with the following minor exceptions. As previously mentioned, instead of using many FRB instances to abort or commit ballots, we use BV to do so. More specifically,

1. When the leader attempts to prepare its candidate ballot, rather than broadcasting $\mathbb{A}$ to all FRB instances (corresponding to ballots) below and incompatible with the candidate, the leader simply requests *prepare*(b) to BV.
2. For a process to determine whether a ballot b is prepared, rather than checking that every FRB below and incompatible with b has delivered $\mathbb{A}$, it receives a *prepared*(b) response from BV.
3. Analogously, rather than broadcasting and delivering $\mathbb{C}$ from an FRB instance, the leader requests *commit* to BV and processes receive *committed* responses from BV.
4. Delay Δ is now set to at least seven rounds of communication delay. In BV, four rounds are needed between a *prepare* (broadcast) request and a *prepared* (deliver) response. Three additional rounds are needed to ensure prepare totality (as described in the proof of Lemma 19).

Next, we prove the correctness of this PBC protocol modulated with BV. We note that the proofs are structurally similar to the correctness proofs of the ABC protocol (Alg. 2).

Lemma 20 *PBC guarantees agreement.*

Proof A well-behaved process decides the value of a ballot after BV responds *committed* for that ballot. By Lemma 16, the commit consistency of BV, no two well-behaved processes issue a *committed* response for incompatible ballots (*i.e.*, ballots with different values), directly implying agreement. □

Lemma 21 *PBC guarantees validity.*

Proof Assuming all processes are well-behaved and that some process has decided a value v, we show that v was proposed by some process. A well-behaved process decides the value of a ballot after BV responds *committed* for that ballot. By Lemma 18, the commit integrity of BV, if a process delivers a *committed* response for a ballot, then some process must have requested *commit* for that ballot. All *commit* requests are from the leader on their own *candidate* ballot.

Next, we claim that the value of every *candidate* ballot can be traced back to the proposal of some process. Consider the *candidate* ballot $b_c = \langle r, v\rangle$ of some well-behaved process p. Value v is either the initial proposal of p or adopted from the value of its *prepared* ballot. The first case directly affirms the claim.

For the second case, v takes on the value of the *prepared* ballot of p. A well-behaved process only ever updates its *prepared* ballot to a ballot b after delivering a *prepared* response for b from BV. By Lemma 17, the prepare integrity of BV, we know that if p delivers a *prepared* response for a ballot, then some process requested *prepare* for that ballot. This process is the leader. Further, the leader requests *prepare* on its *candidate* ballot, meaning v is the value of the *candidate* ballot of yet another leader. Since the number of leaders is finite, by well-founded induction, the second case eventually reduces to the first. □

Lemma 22 *PBC guarantees termination for the set of strongly available processes.*

Proof Similar to the proof of Lemma 11, we consider the system after GST and assume processes no longer time-out. Eventually, a strongly available process p_ℓ is designated as the leader and will lead all strongly available processes through preparation and commitment.

At the start of the round, the leader waits Δ time (seven rounds of communication delay) (L. 28). After GST, this is sufficient time for all messages sent by a well-behaved process in the prepare part of BV to be received and processed. That is, it takes at most seven rounds of communication delay from the time a process requests *prepare* for a ballot, until every strongly available process issues a *prepared* response for that ballot. Then, let b be the largest ballot that has been prepared by any well-behaved process by the end of the Δ wait. By Lemma 19, the prepare totality of BV, since a well-behaved process has issued $prepared(b)$, then every process in P, the set of strongly available processes, also issues $prepared(b)$, including p_ℓ. Thus, after the Δ wait, the leader sets the value of its *candidate* ballot, b_c, to the value of its *prepared* ballot, $b.v$ (L. 33). Then, b_c and b are compatible.

After doing so, p_ℓ requests $prepare(b_c)$ to BV (at L. 36). We claim that (1) every process in P eventually issues $prepared(b_c)$. Then, p_ℓ requests $commit(b_c)$. Upon doing so, every process in P sends $CEcho(b_c)$. By strong availability, every process in P contains a quorum within P, causing every process in P to send $CReady(b_c)$. Using the same reasoning, every process in P gathers a quorum of *CReady* for b_c, leading every such process to issue $committed(b_c)$, after which they decide $b_c.v$ (L. 21).

Then, it remains to prove claim (1). Consider any strongly available process $p \in P$ when it receives $Prepare(b_c)$ from p_ℓ. Specifically consider the condition at L. 27. The first conjunct holds as b_c is the newest ballot, *i.e.*, it has the largest round number since $p_\uparrow$ is the latest leader. For the second conjunct, we show the *ce* variable of p is either (a) compatible with b_c, or (b) $ce = \langle 0, \perp \rangle$.

Recall from above that every process in P, including p, issued $prepared(b)$. Consider the *ce* variable of p right before this happens. There are two cases, depending on the last ballot p sent *CEcho* for. First, notice that p could not have sent *CEcho* for a ballot equal to or above b. This is because to send *CEcho* for a ballot, it must first be prepared (L. 52), and we know b was the largest prepared ballot. Therefore, $ce < b$. We consider two cases:

Case 1: $ce \sim b$. Since $ce \sim b \sim b_c$, (a) holds.

Case 2: $ce \lesssim b$. For p to issue $prepared(b)$, it effectively receives a quorum of $Ready(\mathbb{A})_{\lesssim b}$ (L. 40). By Lemma 14, this same quorum also effectively sends $Ready(\mathbb{A})$ for all ballots below and incompatible with b, including *ce*. Then, the condition on L. 44 is satisfied, in which p resets its *ce* variable to $\langle 0, \perp \rangle$. Thus, (b) holds.

Therefore, we conclude that after p issues $prepared(b)$, the ballot stored in *ce* is either compatible with b_c, or $\langle 0, \perp \rangle$. Therefore, p passes the condition to send $PEcho(b_c)$ at L. 27.

The rest of the proof is straightforward; since p is strongly available, it has a quorum inside P. Thus, p receives $PEcho(b_c)$ from every member of one of its quorums. By Lemma 12, the condition on L. 32 passes, *i.e.*, p effectively received $Echo(\mathbb{A})_{\lesssim b}$ from every member of one of its quorums. Thus, every process in P sends $PReady(b)$ to their followers. By the exact same reasoning, p receives $PReady(b)$ from every member of one of its quorums. By Lemma 12 again, every process in P issues $prepared(b)$, thereby proving claim (1). □

8 Related works

Quorum Systems with Heterogeneous Trust. Ripple [44] and Cobalt [35] pioneered decentralized trust. They let each node specify a list, called the unique node list (UNL), of processes that it trusts. However, they do not consider quorum availability or subsumption.

Stellar [33, 38] presents federated Byzantine quorum systems (FBQS) [24, 25]. where each process p can specify its own quorum slices. A slice of p is a subset of processes that p trusts when they state the same statement. A slice is only a part of a quorum. A quorum is a set of processes that contains a slice for each of its members. A process can construct a quorum starting from one of its own slices, and iteratively probing and including a slice of each process in the set. As processes calculate their own quorums, an HQS is formed. While in an FBQS, each process specifies its slices, in an HQS, each process specifies its quorums.

FBQS motivated the study of HQS, and its requirements for safety and liveness. FBQS and HQS are fundamentally different models although they have similarities to maintain safety and liveness. FBQS considers the intact set, a subset of well-behaved processes with particular conditions. Previous work [24, 34] showed the existence of an intact set even if Byzantine processes lie about their slices. For safety, every pair of quorums should have an intersection in the intact set. For liveness, every intact process should have a quorum inside the intact set. Similarly, for safety, HQS requires quorum intersection at well-behaved processes. For liveness, HQS introduces strong availability, which requires a quorum that is well-behaved and quorum subsuming. At the core of both conditions for liveness is the fact that given a quorum to make progress, each member of the quorum should have a well-behaved quorum to make progress as well. Further research may abstract the similarities into a generalized model. Stellar [24, 34] also presents a federated voting and consensus protocol. The stellar consensus protocol (SCP)

guarantees termination when Byzantine processes stop. In contrast, the consensus protocol in this paper guarantees termination regardless of Byzantine processes.

Similar to this paper, [24] presents a reliable broadcast protocol that is correct even if processes lie about their slices. The validity and totality properties for the reliable broadcast for FBQS are restricted to the intact set. On the other hand, the reliable broadcast protocol in this paper provides totality for all processes that have weak availability, and validity for all processes that have strong availability. Similarly, the conditions of subjective DQS ([24] section 6) are stronger than HQS. For example, subjective DQS requires a correct process to have a failure-prone set that contains all the Byzantine processes; however, in HQS, Byzantine processes can block the quorums of a well-behaved process.

Personal Byzantine quorum systems (PBQS) [34] capture the quorum systems that FBQSs derive from slices, and propose a responsive consensus protocol [1, 3, 43, 48]. They define a notion called quorum sharing which requires quorum subsumption for every quorum. Stellar quorums have quorum sharing if and only if processes do not lie about their slices. (The appendix [31] presents examples.) In this paper, we relax quorum sharing to quorum subsumption, and capture quorums that FBQSs derive even when Byzantine quorums lie about their slices, and show that even if a quorum system does not satisfy quorum sharing, safety can be maintained for all processes, and liveness can be maintained for the set of strongly available processes.

Asymmetric Byzantine quorum systems (ABQS) [4, 15, 16] allow each process to define a subjective dissemination quorum system (DQS), in a globally known system. The followup model [14] lets each process specify a subjective DQS for processes that it knows, transitively relying on the assumptions of other processes. In contrast, HQS lets each process specify its own set of quorums without knowing the quorums of other processes. Further, it does not require the specification of a set of possible Byzantine sets. Further, there are systems where a strongly available set (from HQS) exists but no guild set (from ABQS) exists. (The appendix [31] presents examples.) Therefore, HQS can provide safety and liveness for those executions but ABQS cannot. ABQS presents shared memory and broadcast protocols, and further, rules to compose two ABQSs. On the other hand, this paper proves impossibility results, and presents protocols for reliable broadcast and consensus abstractions. HQS provides strictly stronger guarantees with weaker assumptions. In ABQS, the properties of reliable broadcast are stated for wise processes and the guild. However, this paper states these four properties for well-behaved processes and the strongly available set. Well-behaved processes are a superset of wise processes, and as noted above, in certain executions, the strongly available set is a superset of the guild.

Flexible BFT [36] allows different failure thresholds between learners. Heterogeneous Paxos [45, 46] further generalizes the separation between learners and acceptors with different trust assumptions; it specifies quorums as sets rather than number of processes. These two projects introduce a consensus protocol that guarantees safety or liveness for learners with correct trust assumptions. However, they require the knowledge of all processes in the system. In contrast, HQS only requires partial knowledge of the system, and captures the properties of quorum systems where reliable broadcast and consensus protocols are impossible or possible. Multi-threshold reliable broadcast and consensus [27] and MT-BFT [40] elaborate Bracha [9] to have different fault thresholds for different properties, and different synchrony assumptions. However, they have cardinality-based or uniform quorums across processes. In contrast, HQS supports heterogeneous quorums.

K-consistent reliable broadcast (K-CRB) [7] introduces a relaxed reliable broadcast abstraction where the correct processes can define their own quorum systems. Given a quorum system, it focuses on delivering the smallest number k of different values. In contrast, we propose the weakest condition to solve classical reliable broadcast and consensus. Moreover, K-CRB's relaxed liveness guarantee (accountability) requires public key infrastructure. In contrast, all the results in this paper are for information-theoretic setting.

Our consensus protocol uses eventual leader election. Several other works present view synchronization and eventual leader election for Byzantine replicated systems [10, 11], and dynamic networks [28, 41]. It is interesting to see if their leader election modules can be generalized to the heterogeneous setting, and support responsiveness [5, 48] for our consensus protocol.

Impossibility Results. There are two categories of assumptions about the computational power of Byzantine processes. In the information-theoretic setting, Byzantine process have unlimited computational resources. While in the computational setting, Byzantine processes can not break a polynomial-time bound [23]. In this work, our impossibility results for reliable broadcast and consensus fall in the information-theoretic category. Whether the same results hold in the computational setting is an interesting open question.

FLP [22] proved that consensus is not solvable in asynchronous networks even with one crash failure. Many following works [2, 8, 19, 21, 26, 30] considered solvability, and necessary and sufficient conditions for consensus and reliable broadcast to tolerate f Byzantine failures in partially synchronous networks. The number of processes should be more than $3f$ and the connectivity of the communication graph should be more than $2f$. However, these results apply for cardinality-based quorums, which is a special instance

of HQS. We generalize the reliable broadcast and consensus abstractions to HQS which supports non-uniform quorums, and prove impossibility results for them.

9 Conclusion

This paper presented a general model of heterogeneous quorum systems where each process defines its own set of quorums, and captured their properties. Through indistinguishably arguments, it proved that no deterministic quorum-based protocol can implement the consensus and Byzantine reliable broadcast abstractions on a heterogeneous quorum system that provides only quorum intersection and availability. It introduced the quorum subsumption property, and showed that the three conditions together are sufficient to implement the two abstractions. It presented Byzantine broadcast and consensus protocols for heterogeneous quorum systems, and proved their correctness when the underlying quorum system maintain the three properties.

Appendix A Example execution for consensus

We demonstrate an execution of our consensus protocol, which illustrates its ability to fulfill both safety and liveness properties. Let us assume $\mathcal{P} = \{1, 2, 3, 4\}$, $\mathcal{B} = \{2\}$, $\mathcal{Q}(1) = \{\{1, 2, 3\}\}$, $\mathcal{Q}(3) = \{\{3, 4\}, \{1, 3\}\}$, $\mathcal{Q}(4) = \{\{3, 4\}\}$. All the well-behaved processes are initialized with a default ballot $\langle 0, \perp\rangle$ for their *prepared* and *candidate* variable. The well-behaved processes propose three different values: process 1 propose 3; process 3 propose 5; process 4 propose 2.

In the first round, process 1 is the leader. Upon the proposal at L. 12, all the well-behaved process update their *candidate* with their own proposal value at L. 13. Process 1 aborts all the ballots below and incompatible with its *candidate* through reliable broadcast: it *BCast*$\mathbb{A}$ for instances $\langle 0, \perp\rangle$, $\langle 1, 1\rangle$ and $\langle 1, 2\rangle$ at L. 16. Since the network is partially synchronized, we let the process 4 be partitioned from the rest of processes temporarily. Byzantine process 2 only send *Echo*$\mathbb{A}$ messages for ballot $\langle 0, \perp\rangle$ and $\langle 1, 1\rangle$. Since process 1 has the individual quorum $\{1, 2, 3\}$, it send *Ready*$\mathbb{A}$ messages for the aforementioned two ballots. Process 3 sends *Ready*$\mathbb{A}$ messages for the two ballots, too. We partition process 3 from process 1 and 2 from now on temporarily. Process 1 is able to deliver the $\mathbb{A}$ for ballots $\langle 0, \perp\rangle$ and $\langle 1, 1\rangle$ through BRB at L. 17 and update its *prepared* variable as $\langle 1, 2\rangle$ at L. 18. Since the leader process 1's *prepared* $\neq$ *candidate*, it can not broadcast commit for its *candidate* and the timer triggers.

In the second round, Byzantine process 2 is the leader. When the new leader is elected for a new round, all the well-behaved process increase their round number and update their *candidate*: process 1 updates its *candidate* value to its prepared value 2 at L. 34; process 3 and 4 has not prepared any ballot and update their candidate to their original proposal at L. 32. Then Byzantine leader tries to commit an arbitrary value 4, which no well-behaved process will accept, since the well-behaved process only deliver $\mathbb{C}$ for a BRB instance with the ballot same as its *prepared* at L. 21. We recovery the connection from process 3 and 4 to the rest of the network. Then both process 3 and 4 delivers $\mathbb{A}$ message from previous round at L. 17 and update their *prepared* ballot to $\langle 1, 2\rangle$. No value is decided and the timer trigger again Table 1.

In the third round, well-behaved process 3 is the leader. All the well-behaved process now synchronized to the same highest prepared ballot $\langle 1, 2\rangle$ and update their *candidate* accordingly. Process 3 then abort the rest ballots that are less and incompatible with its *candidate* $= \langle 3, 2\rangle$ through BRB. Byzantine process 2 remain silent in this round to prevent liveness for the weakly available process 1. However, process 3 and 4 are strongly available and is able to successfully deliver the abort messages. Their *prepared* is updated to $\langle 3, 2\rangle$ and process 3 commits this ballot through BRB at L. 20. All the process 3 and 4 deliver the commit message at L. 21 and decide the value 2 in the committed ballot $\langle 3, 2\rangle$ at L. 22.

Appendix B Discussion

An FBQS [24, 38] lets each process p specify its own quorum slices. A slice is a subset of processes that p trusts when they state the same statement. A slice is only a part of a quorum. A quorum is a set of processes that contains a slice for each of its members. A process can construct a quorum starting from one of its own slices, and iteratively probing and including a slice of each process in the set. As each process calculates its own quorums, an HQS is formed.

When Byzantine processes don't lie about their slices, an FBQS enjoys quorum sharing [34]. Consider a quorum q of a process p, and a process p' in it. Since a set is recognized as a quorum only if it contains a slice for each of its members, the process p' has a slice s in q. Processes receive the same set of slices from a given process even if it is Byzantine. If p' starts from s, it can find a quorum that grows no larger than q, *i.e.*, it can stop at q if not earlier. Therefore, q includes a quorum of p'.

However, when Byzantine processes lie about their slices, FBQS does not satisfy quorum sharing. Consider an FBQS with five processes

$$\mathcal{P} = \{1, 2, 3, 4, 5\}$$

and one Byzantine process

$$\mathcal{B} = \{4\}$$

Table 1 An execution for consensus protocol with leader switch

1	2	3	4
$p = \langle 0, \perp \rangle$		$p = \langle 0, \perp \rangle$	$p = \langle 0, \perp \rangle$
$c = \langle 0, \perp \rangle$		$c = \langle 0, \perp \rangle$	$c = \langle 0, \perp \rangle$
Propose(3)		*Propose*(5)	*Propose*(2)
$p = \langle 0, \perp \rangle$		$p = \langle 0, \perp \rangle$	$p = \langle 0, \perp \rangle$
$c = \langle 1, 3 \rangle$		$c = \langle 1, 5 \rangle$	$c = \langle 1, 2 \rangle$
BCast($\langle 0, \perp \rangle$, *Abort*)			
BCast($\langle 1, 1 \rangle$, *Abort*)			
BCast($\langle 1, 2 \rangle$, *Abort*)			
Echo($\langle 0, \perp \rangle$, *Abort*)	*Echo*($\langle 0, \perp \rangle$, *Abort*)	*Echo*($\langle 0, \perp \rangle$, *Abort*)	slow
Echo($\langle 1, 1 \rangle$, *Abort*)	*Echo*($\langle 1, 1 \rangle$, *Abort*)	*Echo*($\langle 1, 1 \rangle$, *Abort*)	
Echo($\langle 1, 2 \rangle$, *Abort*)		*Echo*($\langle 1, 2 \rangle$, *Abort*)	
Ready($\langle 0, \perp \rangle$, *Abort*)	*Ready*($\langle 0, \perp \rangle$, *Abort*)	*Ready*($\langle 0, \perp \rangle$, *Abort*)	slow
Ready($\langle 1, 1 \rangle$, *Abort*)	*Ready*($\langle 1, 1 \rangle$, *Abort*)	*Ready*($\langle 1, 1 \rangle$, *Abort*)	
Deliver($\langle 0, \perp \rangle$, *Abort*)		slow	slow
Deliver($\langle 1, 1 \rangle$, *Abort*)			
$p = \langle 1, 2 \rangle$		$p = \langle 0, \perp \rangle$	$p = \langle 0, \perp \rangle$
$c = \langle 1, 3 \rangle$		$c = \langle 0, \perp \rangle$	$c = \langle 0, \perp \rangle$
time out		time out	time out
$p = \langle 1, 2 \rangle$		$p = \langle 0, \perp \rangle$	$p = \langle 0, \perp \rangle$
$c = \langle 2, 2 \rangle$		$c = \langle 2, 5 \rangle$	$c = \langle 2, 2 \rangle$
	BCast($\langle 2, 4 \rangle$, *Commit*)		
		Deliver($\langle 0, \perp \rangle$, *Abort*)	*Echo*($\langle 0, \perp \rangle$, *Abort*)
		Deliver($\langle 1, 1 \rangle$, *Abort*)	*Echo*($\langle 1, 1 \rangle$, *Abort*)
			Echo($\langle 1, 2 \rangle$, *Abort*)
			Ready($\langle 0, \perp \rangle$, *Abort*)
			Ready($\langle 1, 1 \rangle$, *Abort*)
			Delivery($\langle 0, \perp \rangle$, *Abort*)
			Delivery($\langle 1, 1 \rangle$, *Abort*)
$p = \langle 1, 2 \rangle$		$p = \langle 1, 2 \rangle$	$p = \langle 1, 2 \rangle$
$c = \langle 2, 2 \rangle$		$c = \langle 2, 5 \rangle$	$c = \langle 2, 2 \rangle$
time out		time out	time out
$p = \langle 1, 2 \rangle$		$p = \langle 1, 2 \rangle$	$p = \langle 1, 2 \rangle$
$c = \langle 3, 2 \rangle$		$c = \langle 3, 2 \rangle$	$c = \langle 3, 2 \rangle$
		BCast($\langle 1, 2 \rangle$, *Abort*) ...	
	silent	*Deliver*($\langle 1, 2 \rangle$, *Abort*)...	*Deliver*($\langle 1, 2 \rangle$, *Abort*)...
$p = \langle 1, 2 \rangle$		$p = \langle 3, 2 \rangle$	$p = \langle 3, 2 \rangle$
$c = \langle 3, 2 \rangle$		$c = \langle 3, 2 \rangle$	$c = \langle 3, 2 \rangle$
		BCast($\langle 3, 2 \rangle$, *Commit*)	
	silent	*Deliver*($\langle 3, 2 \rangle$, *Commit*)	*Deliver*($\langle 3, 2 \rangle$, *Commit*)
	silent	*Decide*(2)	*Decide*(2)

The slices of well-behaved processes

$$\mathcal{W} = \{1, 2, 3, 5\}$$

are the following:

$$\S(1) = \{\{1, 2\}\}$$
$$\S(2) = \{\{2, 3\}, \{2, 4\}, \{2, 5\}\}$$
$$\S(3) = \{\{3, 4\}\}$$
$$\S(5) = \{\{2, 5\}\}$$

However, the Byzantine process 4 provides different quorum slices to different well-behaved processes. The slices that it sends to each other process are the following:

$$\S(4)_1 = \{\{1, 4\}\}$$
$$\S(4)_2 = \{\{3, 4\}\}$$
$$\S(4)_3 = \{\{2, 4\}\}$$
$$\S(4)_5 = \{\{4, 5\}\}$$

By the definition of quorums in FBQS, we have the following individual quorums for each well-behaved process:

$$\mathcal{Q}(1) = \{\{1, 2, 4\}, \{1, 2, 5\}\}$$
$$\mathcal{Q}(2) = \{\{2, 3, 4\}, \{2, 5\}\}$$
$$\mathcal{Q}(3) = \{\{2, 3, 4\}\}$$
$$\mathcal{Q}(5) = \{\{2, 5\}\}$$

We can see that $\mathcal{Q}$ does not have quorum sharing, since the quorum $\{1, 2, 4\}$ does not include any quorum for process 2. However, $\mathcal{Q}$ is strongly available for processes $\{1, 2, 5\}$, since they all have quorums that are well-behaved and quorum subsuming.

Abstract quorums from PBQS are over-approximations of FBQS satisfy quorum-sharing [34]: a set q is an abstract quorum of a process p if p is Byzantine, or every well-behaved member of q (including p) has a slice contained in q. We note that abstract quorums satisfy quorum-sharing. However, as shown in our example, not all abstract quorums of a process can be observed by that process when Byzantine processes lie about their quorum slices. For example, $q = \{1, 2, 4\}$ is an abstract quorum of process 2 since the slice $\{2, 4\}$ of 2 is in it, and the slice $\{1, 2\}$ for 1 is in it. However, as we showed above, 2 does not calculate q as a quorum when the Byzantine process 4 lies. Therefore, relaxation of the quorum sharing is necessary when analyzing quorum systems such as FBQS when Byzantine processes lie.

Let us now consider ABQS with the notion of guild set. There are systems where a strongly available set exists but no guild exists. Therefore, HQS can provide safety and liveness for those executions but ABQS cannot. We show a quorum system that has an empty guild but has non-empty strongly available set and quorum intersection. Let

$$\mathcal{P} = \{1, 2, 3, 4\}$$

and process

$$\mathcal{B} = \{3\}$$

is Byzantine. Let the asymmetric failure prone system be

$$\mathcal{F}(1) = \{\{3, 4\}, \{2\}\}$$
$$\mathcal{F}(2) = \{\{3, 4\}, \{1\}\}$$
$$\mathcal{F}(3) = \{\{1, 2\}\}$$
$$\mathcal{F}(4) = \{\{2, 3\}\}$$

The canonical quorum systems are

$$\mathcal{Q}(1) = \{\{1, 2\}, \{1, 3, 4\}\}$$
$$\mathcal{Q}(2) = \{\{1, 2\}, \{2, 3, 4\}\}$$
$$\mathcal{Q}(3) = \{\{3, 4\}\}$$
$$\mathcal{Q}(4) = \{\{1, 4\}\}$$

This HQS has quorum intersection and a non-empty strongly available set: all the quorums of well-behaved processes have at least one correct process in their intersection. The set $\{1, 2\}$ is a strongly available set for $\mathcal{Q}$. However, it is not an ABQS with generalized B^3 condition. For processes 1 and 2, $\mathcal{F}_1 = \{2\}, \mathcal{F}_2 = \{1\}$ and $\mathcal{F}_{12} = \{3, 4\}$, and we have $\mathcal{P} \subseteq \{2\} \cup \{1\} \cup \{3, 4\}$, which violates generalized B^3. Therefore this quorum system is not an ABQS. There is no guild set.

Acknowledgements We would like to thank DISC '23 reviewers for detailed and constructive reviewers. Further, we would like to specially thank Giuliano Losa for his insightful comments.

References

1. Abraham, I., Stern, G.: Information theoretic hotstuff. (2020) arXiv preprint arXiv:2009.12828
2. Aguilera, M.K., Delporte-Gallet, C., Fauconnier, H., Toueg, S.: Consensus with byzantine failures and little system synchrony. In International Conference on Dependable Systems and Networks (DSN'06), pages 147–155. IEEE, (2006)
3. Alistarh, D., Aspnes, J., Ellen, F., Gelashvili, R., Zhu, L.: Why extension-based proofs fail. In Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, pages 986–996 (2019)
4. Alpos, O., Cachin, C., Zanolini, L.: How to trust strangers: Composition of byzantine quorum systems. In 2021 40th International Symposium on Reliable Distributed Systems (SRDS), pages 120–131. IEEE, (2021)
5. Attiya, H., Dwork, C., Lynch, N., Stockmeyer, L.: Bounds on the time to reach agreement in the presence of timing uncertainty. Journal of the ACM (JACM) **41**(1), 122–152 (1994)
6. Baudet, M., Ching, A., Chursin, A., Danezis, G., Garillot, F., Li, Z., Malkhi, D., Naor, O., Perelman, D., Sonnino, A.: State machine replication in the libra blockchain. The Libra Assn. Tech. Rep **7**, (2019)
7. Bezerra, J.P., Kuznetsov, P., Koroleva, A.: Relaxed reliable broadcast for decentralized trust. In Networked Systems: 10th International Conference, NETYS 2022, Virtual Event, May 17–19, 2022, Proceedings, pages 104–118. Springer, (2022)
8. Borcherding, M.: Levels of authentication in distributed agreement. In International Workshop on Distributed Algorithms, pages 40–55. Springer, (1996)
9. Bracha, G., Toueg, S.: Asynchronous consensus and broadcast protocols. Journal of the ACM (JACM) **32**(4), 824–840 (1985)
10. Bravo, M., Chockler, G., Gotsman, A.: Liveness and latency of byzantine state-machine replication. In 36th International Symposium on Distributed Computing (DISC 2022). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2022)
11. Bravo, M., Chockler, G., Gotsman, A.: Making byzantine consensus live. Distrib. Comput. **35**(6), 503–532 (2022)
12. Buchman, E.: Tendermint: Byzantine fault tolerance in the age of blockchains. PhD thesis, University of Guelph, (2016)
13. Buchman, E., Guerraoui, R., Komatovic, J., Milosevic, Z., Seredinschi, D.A., Widder, J.: Revisiting tendermint: Design tradeoffs, accountability, and practical use. In 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S), pages 11–14. IEEE, (2022)
14. Cachin, C., Losa, G., Zanolini, L.: Quorum systems in permissionless network. arXiv preprint arXiv:2211.05630, (2022)
15. Cachin, C., Tackmann, B.: Asymmetric distributed trust. In 23rd International Conference on Principles of Distributed Systems (OPODIS 2019). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2020)
16. Cachin, C., Zanolini, L.: From symmetric to asymmetric asynchronous byzantine consensus. arXiv preprint arXiv:2005.08795, (2020)
17. Carr, H., Jenkins, C., Moir, M., Miraldo, V.C., Silva, L.: Towards formal verification of hotstuff-based byzantine fault tolerant consensus in agda. In NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings, pages 616–635. Springer, (2022)
18. Castro, M., Liskov, B., et al.: Practical byzantine fault tolerance. In OSDI **99**, 173–186 (1999)
19. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Hadzilacos, V., Kouznetsov, P., Toueg, S.: The weakest failure detectors to solve certain fundamental problems in distributed computing. In Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing, pages 338–346 (2004)
20. Dwork, C., Lynch, N., Stockmeyer, L.: Consensus in the presence of partial synchrony. Journal of the ACM (JACM) **35**(2), 288–323 (1988)
21. Fischer, M.J., Lynch, N.A., Merritt, M.: Easy impossibility proofs for distributed consensus problems. Distrib. Comput. **1**(1), 26–39 (1986)
22. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM (JACM) **32**(2), 374–382 (1985)
23. Garay, J., Kiayias, A.: Sok: A consensus taxonomy in the blockchain era. In Cryptographers' track at the RSA conference pages 284–318. Springer, (2020)
24. García-Pérez, Á., Gotsman, A.: Federated byzantine quorum systems. In 22nd International Conference on Principles of Distributed Systems (OPODIS 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2018)
25. García-Pérez, Á., Schett, M.A.: Deconstructing stellar consensus (extended version). arXiv preprint arXiv:1911.05145, (2019)
26. Goren, G., Moses, Y., Spiegelman, A.: Probabilistic indistinguishability and the quality of validity in byzantine agreement. arXiv preprint arXiv:2011.04719, (2020)
27. Hirt, M., Kastrati, A., Liu-Zhang, C.-D.: Multi-threshold asynchronous reliable broadcast and consensus. Cryptology ePrint Archive (2020)
28. Ingram, R., Shields, P., Walter, J.E., Welch, J.L.: An asynchronous leader election algorithm for dynamic networks. In 2009 IEEE International Symposium on Parallel & Distributed Processing, pages 1–12. IEEE, (2009)
29. Lamport, L.: Lower bounds for asynchronous consensus. Distrib. Comput. **19**, 104–125 (2006)
30. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Trans. Program. Lang. Syst. pages 382–401 (1982)
31. Li, X., Chan, E., Lesani, M.: Quorum subsumption for heterogeneous quorum systems. technical report. In International Symposium on Distributed Computing (DISC 2023), (2023)
32. Li, X., Lesani, M.: Open heterogeneous quorum systems, (2023)
33. Lokhava, M., Losa, G., Mazières, D., Hoare, G., Barry, N., Gafni, E., Jove, J., Malinowsky, R., McCaleb, J.: Fast and secure global payments with stellar. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 80–96 (2019)
34. Losa, G., Gafni, E., Mazieres, D.: Stellar consensus by instantiation. In 33rd International Symposium on Distributed Computing (DISC 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, (2019)
35. MacBrough, E.: Cobalt: Bft governance in open networks. arXiv preprint arXiv:1802.07240, (2018)
36. Malkhi, D., Nayak, K., Ren, L.: Flexible byzantine fault tolerance. In Proceedings of the 2019 ACM SIGSAC conference on computer and communications security, pages 1041–1053 (2019)
37. Malkhi, D., Reiter, M.: Byzantine quorum systems. Distributed computing **11**(4), 203–213 (1998)
38. Mazieres, D.: The stellar consensus protocol: A federated model for internet-level consensus. Stellar Development Foundation **32**, 1–45 (2015)
39. Andrew Miller, Yu., Xia, K.C., Shi, E., Song, D.: The honey badger of bft protocols. In Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, pages 31–42 (2016)
40. Momose, A., Ren, L.: Multi-threshold byzantine fault tolerance. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pages 1686–1699 (2021)
41. Mostefaoui, A., Raynal, M., Travers, C., Patterson, S., Agrawal, D., Abbadi, A.E.L.: From static distributed systems to dynamic

systems. In 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05), pages 109–118. IEEE, (2005)
42. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. White paper, (2008)
43. Pass, R., Shi, E.: Thunderella: Blockchains with optimistic instant confirmation. In Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part II 37, pages 3–33. Springer, (2018)
44. Schwartz, D., Youngs, N., Britto, A.: The ripple protocol consensus algorithm. Ripple Labs Inc White Paper **5**(8), 151 (2014)
45. Sheff, I., Wang, X., van Renesse, R.: and Andrew C Myers. Heterogeneous paxos. In OPODIS: International Conference on Principles of Distributed Systems, number 2020 in OPODIS (2021)
46. Sheff, I.C., van Renesse, R., Myers, A.C.: Distributed protocols and heterogeneous trust: Technical report. arXiv preprint arXiv:1412.3136, (2014)
47. Veronese, G.S., Correia, M., Bessani, A.N., Lung, L.C., Verissimo, P.: Efficient byzantine fault-tolerance. IEEE Trans. Comput. **62**(1), 16–30 (2011)
48. Yin, M., Malkhi, D., Reiter, M.K., Gueta, G.G., Abraham, I.: Hotstuff: BFT consensus with linearity and responsiveness. In Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing, pages 347–356, (2019)