Quorum Subsumption for Heterogeneous Quorum Systems

Xiao Li
University of California, Riverside, USA

Eric Chan
University of California, Riverside, USA

Mohsen Lesani
University of California, Riverside, USA

Abstract

Byzantine quorum systems provide higher throughput than proof-of-work and incur modest energy consumption. Further, their modern incarnations incorporate personalized and heterogeneous trust. Thus, they are emerging as an appealing candidate for global financial infrastructure. However, since their quorums are not uniform across processes anymore, the properties that they should maintain to support abstractions such as reliable broadcast and consensus are not well-understood. It has been shown that the two properties quorum intersection and availability are necessary. In this paper, we prove that they are not sufficient. We then define the notion of quorum subsumption, and show that the three conditions together are sufficient: we present reliable broadcast and consensus protocols, and prove their correctness for quorum systems that provide the three properties.

2012 ACM Subject Classification Computing methodologies → Distributed algorithms; Computer systems organization → Availability; Computer systems organization → Reliability

Keywords and phrases Distributed Systems, Impossibility Results, Byzantine fault tolerance

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.16

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 have always had modest energy consumption. Further, since its advent as PBFT [18], many recent extensions [47, 39, 48, 17, 6, 12, 13] 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 [38, 33] emerged to bring *heterogeneity* and openness to Byzantine quorum systems. They let every process declare its own set of quorums, or the processes it trusts 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 quorum intersection and weak availability properties are necessary for the quorum system to implement the consensus abstraction. Quorum intersection requires that every pair of quorums overlap at a well-behaved process. The safety of consensus relies on the quorum intersection property of the underlying quorum system: intuitively, if an operation communicates with a quorum, and a later operation communicates with another quorum, a single well-behaved process in their intersection can make 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, the quorum system is available to that process through that quorum. Since a process needs to communicate with at least one quorum to terminate, the liveness properties are dependent on the availability of the quorum system.

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 safety or liveness properties. 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 every quorum is uniformly a quorum of all its members. However, in general, it does not hold for heterogeneous quorum systems. Previous work showed that it also holds for Stellar quorums if Byzantine processes do not lie about their slices.

Since Byzantine processes' quorums is arbitrary, in practice, quorum sharing is too strong. 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 properties.

In summary, this paper makes the following contributions.

- Properties of quorum-based protocols (Section 3) and specifications of reliable broadcast and consensus on heterogeneous quorum systems (Section 4).
- Proof of insufficiency of quorum intersection and availability to solve consensus (Subsection 5.1) and reliable broadcast (Subsection 5.2).
- Sufficiency of quorum intersection, quorum availability and quorum subsumption to solve consensus and reliable broadcast. We present protocols for reliable broadcast (Subsection 6.1) and consensus (Subsection 6.2), and their proofs of correctness.

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 theorems and protocols in the 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. A quorum system is hosted on a set of processes $\mathcal{P}$. For every execution, we can partition the set $\mathcal{P}$ into *Byzantine* $\mathcal{B}$ and *well-behaved* $\mathcal{W} = \mathcal{P} \setminus \mathcal{B}$ processes. Well-behaved processes follow the given protocol, while Byzantine processes can deviate from the protocol arbitrarily.

We assume that the network is partially synchronized, *i.e.*, after an unknown global stabilization time (GST), if both the sender and receiver are well-behaved, the message will eventually be delivered with a known bounded delay [20].

Heterogeneous Quorum Systems (HQS). To represent subjective trust, we let each process specify its own quorums. A quorum q of process p is a non-empty subset P of $\mathcal{P}$ that p trusts to get information from if it obtains the same information from each member of P. (In practice, a quorum of p can contain p itself, although the model does not require it.) Each process p stores its own set of quorums that we call individual quorums of p. Any superset of a quorum of p is also a quorum of p; thus, there are minimal quorums: a quorum of p is a minimal quorum of p if none of its strict subsets is a quorum of p. Thus, to avoid redundancy, p can ignore its quorums that are proper supersets of its minimal quorums. Thus, each process stores only its individual minimal quorums.

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

Since the trust assumptions of Byzantine processes can be arbitrary, their quorums can be left unspecified. Figure 1 presents an example quorum system. When obvious from the context, we say quorums of p to refer to the individual minimal quorums of p, and use $\mathcal{Q}$ to refer to the set of all individual minimal quorums of the system, i.e. the co-domain of $\mathcal{Q}$. Additionally, we say quorum systems to refer to heterogeneous quorum systems. A process p is a *follower* of a process p' iff there is a quorum $q \in \mathcal{Q}(p)$ that includes p'.

In dissemination quorum system (DQS) [37] (and the cardinality-based quorum systems as a special case), quorums are uniform for all processes. Processes have the same set of individual minimal 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.

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 on 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 a 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 the running example in Figure 1, all the quorums of well-behaved processes intersect at at least one of well-behaved processes in $\{1, 3, 4\}$.

▶ **Definition 2** (Quorum Intersection). *A quorum system $\mathcal{Q}$ has quorum intersection iff 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. We say that 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 will define.) 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 it requires only a subset and not necessarily all well-behaved processes to have a well-behaved quorum. In Figure 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. Each process in that subset can always communicate with a quorum independently of Byzantine processes.

▶ **Definition 3** (Weak Availability). *A quorum system is weakly available for a set of processes P iff every process in P has at least one quorum that is a subset of well-behaved processes $\mathcal{W}$. A quorum system is available iff it is available for a non-empty set of processes.*

If a quorum system is weakly available, there is at least one well-behaved process that can communicate with a quorum independently of Byzantine 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 4** (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 cardinality-based quorum systems where the system contains $3f + 1$ processes. Any set of size $f + 1$ is a blocking set for all well-behaved processes, since a set with $f + 1$ processes intersects with any quorum, a set with $2f+1$ processes. In Figure 1, well-behaved process 5 is blocked by $\{2\}$, since its only quorum $\{1, 2, 3, 5\}$ intersect with $\{2\}$

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

Figure 1 Quorum System Example

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 (Section 6). For now, we prove a lemma for blocking sets. In order to state the lemma, we generalize the notion of availability. Given a set of processes P, we generalize availability for P at the complete set of well-behaved processes $\mathcal{W}$ (Definition 3) to availability for P at a subset P' of well-behaved processes. We say that a quorum system is weakly available for a set of processes P at a subset of well-behaved processes P' iff every process in P has at least one quorum that is a subset of P'.

▶ **Lemma 5.** *In every quorum system that is weakly available for a set of processes P at P', every blocking set of every process in P intersects P'.*

Proof. Consider a quorum system that is weakly available for P at P', a process p in P, and a set of processes P'' that blocks p. By the definition of available, there is at least one quorum q of p that is a subset of P'. By the definition of blocking set (Definition 4), q intersects with P''. Hence, P' intersects P'' as well. ◀

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 iff every process in q has a quorum that is included in q, i.e., $\forall p \in q.\ \exists q' \in \mathcal{Q}(p).\ q' \subseteq q$. We say that $\mathcal{Q}$ is quorum subsuming for a set of quorums if it is quorum subsuming for each quorum in the set.*

In Figure 1, $\mathcal{Q}$ is quorum subsuming for $\{3, 4\}$: both members in this quorum have the quorum $\{3, 4\}$ that is trivially a subset of itself. However, $\mathcal{Q}$ is not quorum subsuming for process 1's quorum $\{1, 4\}$: process 4's only quorum $\{3, 4\}$ is not a subset of $\{1, 4\}$.

Quorum subsumption is inspired by and weakens the notion of quorum sharing [34]. Quorum sharing requires the above subsumption property for all quorums. Thus, many quorum systems including Ripple and Stellar do not satisfy it (unless Byzantine processes do not lie about their slices [34].) They can maintain the subsumption property only for quorums of a well-behaved subset of processes. In particular, no requirement can be made for quorums of Byzantine 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.

sender	1	2	3	4
$BCast(m_1)$				
	$Echo(m_1)$		$Echo(m_1)$	$Echo(m_1)$
		$Ready(m_2)$		
	$Ready(m_1)$		$Ready(m_2)$	$Ready(m_2)$
	blocked forever			$Deliver(m_2)$

Table 1 Non-termination for Bracha protocol with blocking sets

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, or communicate with a subset of processes that contains at least one well-behaved process. Let us see intuitively how quorum subsumption can support liveness properties. Consider a quorum system $\mathcal{Q}$ for processes $\mathcal{P} = \{1, 2, 3, 4\}$ where the Byzantine processes are $\{2\}$, and $\mathcal{Q}(1) = \{\{1, 3, 4\}\}$, $\mathcal{Q}(3) = \{\{1, 2, 3\}\}$, and $\mathcal{Q}(4) = \{\{2, 3, 4\}\}$. The quorum system $\mathcal{Q}$ has quorum intersection, and is weakly available for the set $\{1\}$ since there is a well-behaved quorum $\{1, 3, 4\}$ for the process 1. In the classic Bracha protocol, the sender broadcasts $BCast(m)$, a well-behaved broadcasts $Echo(m)$ when it receives it from the sender, it broadcasts $Ready(m)$ after receiving $2f + 1$ $Echo(m)$ or $f + 1$ $Ready(m)$ messages, and finally, delivers m if it receives $2f + 1$ $Ready(m)$ messages. In Stellar [33] and follow-up works [34, 24, 15], the check for receiving $Ready(m)$ messages from $f + 1$ processes is replaced with receiving $Ready(m)$ messages from a blocking set of the current process. Let's consider the example execution presented in Table 1; it gives an intuition of why the quorum system needs stronger conditions than weak availability. Consider a Byzantine sender who sends $BCast(m_1)$ to process $\{1, 3, 4\}$. Well-behaved processes $\{1, 3, 4\}$ send out $Echo(m_1)$ to each other. We let process 1 deliver $Echo(m_1)$ messages from process 1, 3, and 4 first; it then sends out $Ready(m_1)$ messages. We note that the two processes 3, and 4 cannot broadcast $Ready(m_1)$ since they have not received $Echo(m_1)$ from a quorum of their own. Then the Byzantine process 2 sends $Ready(m_2)$ messages to process $\{3, 4\}$. Since the set $\{2\}$ is blocking for the quorums of both processes 3 and 4, both send out $Ready(m_2)$ messages. These broadcast protocols prevent a process that is ready for a value from getting ready for another value. Therefore, although $\{3\}$ and $\{4\}$ are both blocking sets for the process 1, it cannot become ready for m_2. Process 1 never receives enough *Ready* messages for either m_1 or m_2 to deliver a message, and is blocked forever. If the quorum $\{1,3,4\}$ for 1 had the quorum subsumption property, then 3 and 4 could send out $Ready(m_1)$ messages, and eventually 1 would make progress.

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 7** (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 previous running example Figure 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 8** (Strong Availability). *A quorum system $\mathcal{Q}$ has strong availability for a subset of processes P iff every process in P has at least one complete quorum. We call P a* strongly available set *for $\mathcal{Q}$, and call 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 his behalf in the face of Byzantine attacks. In Figure 1, $\mathcal{Q}$ is strongly available for $\{3,4\}$. In contrast, $\mathcal{Q}$ is only weakly available for process 1, since its quorum $\{1,2,3\}$ includes 2 that is not well-behaved, and its other quorum $\{1,4\}$ is well-behaved but not a complete quorum.

By Lemma 5, every blocking set of every strongly available process contains at least one well-behaved process.

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 9** (Determinism). *A protocol is deterministic iff 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 [34, 29] 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 needs 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.

A process's local state is only affected by the information that it receives from the members of it's quorums.

▶ **Axiom 3** (Locality). *The state of a well-behaved process changes upon receiving a message only if the sender is a member of one of its quorums.*

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 system), and issues delivery responses.

▶ **Definition 10** (Specification of Reliable Broadcast).

- *(Validity for a set of well-behaved processes P). If a well-behaved process p 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 p, then m was previously broadcast by p.*
- *(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 called federated voting. Similar to reliable broadcast, the abstraction accepts a broadcast request from processes, and issues delivery responses. In contrast to reliable broadcast where there is a dedicated sender, in federated voting, every process can broadcast a message. The specification of federated voting is similar to that of reliable broadcast except for validity. The messages that well-behaved processes broadcast may not be the same. Therefore, the validity property provides guarantees only when the messages are the same or there is only one sender. The validity property for a set of well-behaved processes P guarantees that if all well-behaved processes broadcast a message m, or only one well-behaved process broadcasts a message m, then every process in P eventually 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 11** (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 the information-theoretic settings (Fault axiom [21]), where byzantine processes have unlimited computational power, and can show arbitrary behavior. However, processes communicate only over secure channels so that the recipient knows the identity of the sender. A Byzantine process is unable to impersonate a well-behaved process. This is similar to the classic unauthenticated Byzantine general problem [30], and is necessary for open decentralized blockchains and HQS, where the trusted authorities including public key infrastructures may not be available.

The two proofs will take a similar approach. First, we assume there does exist a protocol for our distributed abstraction that satisfies all the desired specifications. We then present a quorum system $\mathcal{Q}$ and consider its executions that have quorum intersection and availability in the face of Byzantine attacks. We then show through a series of indistinguishable executions that the 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 not able 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.

We consider binary proposals for consensus, and binary values (from the sender) for reliable broadcast. For the consensus abstraction, we succinctly present the values that processes propose as as a vector of values that we call a configuration. If the initial value of a process is $\perp$ in the configuration, that process is considered Byzantine. Otherwise, the process is well-behaved. For example, a configuration $C = \langle 0, 0, \perp \rangle$ denotes the first and second process proposing zero and the third process being Byzantine.

5.1 Consensus

We first consider consensus protocols in HQS.

▶ **Theorem 12.** *Quorum intersection and weak availability are not sufficient for deterministic quorum-based consensus protocols to provide validity, agreement and termination for weakly available processes.*

Proof. We suppose there is a quorum-based consensus protocol that guarantees validity, agreement, and termination for every quorum system $\mathcal{Q}$ with quorum intersection and weak availability, towards contradiction. 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 $\mathcal{Q}$ has 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 Figure 2: $C_0 = \langle 0, 0, 0\rangle$, $C_1 = \langle 1, 1, 1\rangle$, $C_2 = \langle 0, 1, \perp\rangle$, and $C_3 = \langle \perp, 1, 1\rangle$. The goal is now to show a series of executions over the configurations so that at least one property of the protocol is violated.

- We begin with execution E_0 (shown in red) with the initial configuration C_0. 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 can reach this decision with only processes $\{a, c\}$ taking steps.
- Next, we have execution E_1 (shown in blue) with initial configuration C_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.
- Next, we have execution E_2 as a sequence of E_1 and E_0, with initial configuration C_2. Suppose messages 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. This cause process a to decide 0. Now let Byzantine process c stay silent, and messages between processes a and b be delivered. By termination for b, agreement and quorum sufficiency, process a makes b decide 0 as well (shown in green).
- Lastly, we have execution E_3 with initial configuration C_3. Suppose messages between b to c are delivered in the beginning. We let processes $\{b, c\}$ replay E_1; thus, c decides 1. Then, Byzantine process a sends messages to b as if it were at the end of E_2. In turn, b decides 0. Thus, agreement is violated as two well-behaved processes decided differently. ◀

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 process, 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 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.

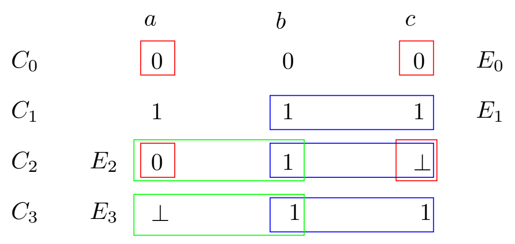

Figure 2 Indistinguishable Executions

5.2 Byzantine Reliable Broadcast

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

For the reliable broadcast abstraction, 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 are not able to distinguish executions based on the sender.

▶ **Theorem 13.** *Quorum intersection and weak availability are not sufficient for deterministic quorum-based reliable broadcast protocols to provide validity and totality for weakly available processes, and consistency.*

Proof. The proof is similar to the proof for consensus. In fact, we will reuse the construction. There are differences between reliable broadcast and consensus specifications in (1) their validity properties, and (2) their totality and termination properties respectively. The proof can be adjusted for these differences. For reliable broadcast, we need a sender process s who broadcasts a message. In executions that we want a well-behaved process to deliver the message m, we either (1) keep the sender s well-behaved and have it send m, and 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.

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 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 1, and messages between processes b and c are delivered. By validity for weakly available processes, and quorum sufficiency, process c delivers 1, 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. 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 0, then process b will 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

Algorithm 1 Byzantine Reliable Broadcast (BRB)

```
1  Implements: ReliableBroadcast
2      request : $broadcast(v)$
3      response : $deliver(v)$
4  Vars:
5      $Q$                                  ▷ Minimal quorums of self
6      $\mathcal{F}$ : Set[$\mathcal{P}$]                       ▷ The followers of self
7      $echoed, readied, delivered$ : Boolean ← false
8      $E, R : V \mapsto$ Set[$\mathcal{P}$] ← $\emptyset$
                           ▷ Set of echoed and readied processes
9  Uses:
10     $ptp$ : PointToPointLink
11 upon request $broadcast(v)$ from sender
12     $ptp$ request $send(p, BCast(v))$ for each $p \in \mathcal{P}$
13 upon $ptp$ response $deliver(p', BCast(v))$
14     if $\neg echoed$ then
15         $echoed$ ← true
16         $ptp$ request $send(p, Echo(v))$ for each $p \in \mathcal{F}$
17 upon $ptp$ response $deliver(p', Echo(v))$
18     $E(v) \leftarrow E(v) \cup \{p'\}$
19     if $\neg readied \wedge \exists q \in Q.\ q \subseteq E(v)$ then
20         $readied$ ← true
21         $ptp$ request $send(p, Ready(v))$ for each $p \in \mathcal{F}$
22 upon $ptp$ response $deliver(p', Ready(v))$
23     $R(v) \leftarrow R(v) \cup \{p'\}$
24     if $\neg readied \wedge R$ is a blocking set of self then
25         $readied$ ← true
26         $ptp$ request $send(p, Ready(v))$ for each $p \in \mathcal{F}$
27     if $\neg delivered \wedge \exists q \in Q.\ q \subseteq R(v)$ then
28         $delivered$ ← true
29         response $deliver(v)$
```

and c replay E_1. Thus, process c delivers 1. Afterwards, messages between process b and c are delayed, and the Byzantine process a replays E_2. Again, 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 will deliver 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. Now, we show that 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 Algorithm 1, we adapt the Bracha protocol [9] to show that quorum intersection and strong availability together are sufficient for Byzantine reliable broadcast. The parts that are different from the classical protocol are highlighted in blue.

Each process stores the set of its individual minimal quorums Q, and its set of followers $\mathcal{F}$. It also stores the boolean flags *echoed*, *readied*, and *delivered* which record actions the process has taken to avoid duplicate actions. It further uses point-to-point links *ptp* to each of its followers. Upon receiving a request to broadcast a value v (at L. 11), the sender broadcasts the value v to all processes (at L. 12). Upon receiving the message from the sender (at L. 13), a well-behaved process echoes the message among its followers (at L. 16) only if it has not already *echoed*. When a well-behaved process receives a quorum of consistent echo messages (at L. 17), it sends ready messages to all its followers (at L. 21). A well-behaved process can also send a ready message when it receives consistent ready messages from a blocking set (at L. 24). When a well-behaved process receives a quorum of consistent ready messages for v

(at L. 27), it delivers v (at L. 29). The implementation of the federated voting abstraction is similar. The only difference is that there can be multiple senders (at L. 11).

We prove that this protocol implements Byzantine reliable broadcast when the quorum system satisfies quorum intersection, and strong availability. We remember that strong availability requires both weak availability and quorum subsumption. More precisely, it requires a well-behaved quorum q for a process p, and quorum subsumption for q.

▶ **Theorem 14.** *Quorum intersection and strong availability are sufficient to implement Byzantine reliable broadcast.*

This theorem follows from five lemmas in the appendix [31] that prove the protocol satisfies the specification of Byzantine reliable broadcast that we defined in Definition 10. Consider a quorum system with quorum intersection, and strong availability for P. Here, we state and prove only the validity property.

▶ **Lemma 15.** *The BRB protocol guarantees validity for P.*

Proof. Consider a well-behaved sender that broadcasts a message m. We show that every process in P eventually delivers m. By availability, every process $p \in P$ has a complete quorum q. Consider a process $p' \in q$. By quorum subsumption, p' has a quorum $q' \subseteq q$. By availability, all members of q (including q') are well-behaved. Thus, when they receive m from the sender, they all echo it to their followers. The processes in q' have p' as a follower. Thus, p' receives consistent echo messages for m from one of its quorums q'. Thus, p' sends out ready messages for m to its followers. Thus, all processes in q send out ready messages for m to their followers. The processes in q have p as a follower. Therefore, p receives a quorum of consistent ready messages for m from one of its quorums q, and delivers m. ◀

6.2 Byzantine Consensus Protocol

In this section, we show that quorum intersection and strong availability are sufficient to implement Byzantine consensus. We first present the consensus protocol for heterogeneous quorum systems, and then prove its correctness.

At a high level, the protocol proceeds in rounds with assigned leaders for each. Ballots that carry proposal values are totally ordered. A leader tries to commit its own candidate ballot only after aborting any lower ballot in the system. Leaders use the federated voting abstraction (that we saw in Section 4) to abort or commit ballots. There may be multiple leaders or Byzantine leaders before GST, and they may broadcast contradicting abort and commit messages for the same ballot. However, by the consistency property of federated voting, 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. Ballots are totally ordered by first their round numbers, and then their values: a ballot $\langle r, v\rangle$ is below another $\langle r', v'\rangle$, written as $\langle r, v\rangle < \langle r', v'\rangle$, if $r < r'$ or $r = r' \wedge v < v'$. Two ballots $b = \langle r, v\rangle$ and $b' = \langle r', v'\rangle$ are compatible, $b \sim b'$, if they have the same value, *i.e.*, $v = v'$; otherwise, they are incompatible, $b \not\sim b'$. We say that a ballot is below and incompatible with another, $b \lnsim b'$, if $b < b'$ and $b \not\sim b'$. For message passing communication, we assume batched network semantics (BNS), 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 the validity property can be violated.)

The protocol is similar to SCP [38, 25] in structure; the important difference is that this protocol uses leaders [34] and guarantees termination. Our protocol guarantees termination

Algorithm 2 Byzantine Consensus

```
1  Implements: Consensus
2    request : $propose(v)$
3    response : $decide(v)$
4  Vars:
5    $round : \mathbb{N}^+ \leftarrow 0$    ▷ Current round number
6    $candidate, prepared : \langle \mathbb{N}^+, V \rangle \leftarrow \langle 0, \bot \rangle$
7    $leader : \mathcal{P} \leftarrow p_0$    ▷ current leader
8  Uses:
9    $fv : B \mapsto$ ByzantineReliableBroadcast
10   $le$ : EventualLeaderElection
11 upon request $propose(v)$
12     $candidate \leftarrow \langle 1, v \rangle$
13     if self $= leader$ then
14         $fv(b')$ request $broadcast(\mathbb{A})$ for all
             $b' \lnsim candidate$
15 upon $fv(b')$ response $deliver(p, \mathbb{A})$ for all
     $b' \lnsim b$ where $prepared < b$
16     $prepared \leftarrow b$
17     if self $= leader \wedge prepared = candidate$
        then
18         $fv(candidate)$ request $broadcast(\mathbb{C})$
19 upon $fv(b)$ response $deliver(p, \mathbb{C})$ where
     $b = prepared \wedge p = leader$
20     response $decide(b.v)$
21 upon $timeout$ triggered
22     $le$ request $Complain(round)$
23 upon $le$ response $new\text{-}leader(p)$
24     $leader \leftarrow p$
25     $round \leftarrow round + 1$
26     if self $= leader$ then
27         Delay for time $\Delta$
28     start-timer$(round)$
29     if $prepared = \langle 0, \bot \rangle$ then
30         $candidate \leftarrow \langle round, candidate.v \rangle$
31     else
32         $candidate \leftarrow \langle round, prepared.v \rangle$
33     if self $= leader$ then
34         $fv(b')$ request $broadcast(\mathbb{A})$ for all
             $b' \lnsim candidate$
```

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 [38, 24] has not yet decided in some execution, then for every continuation of that execution in which all the Byzantine processes stop, the process p eventually decides.)

Each process stores four local variables: *round* is the current round number, *candidate* is the ballot that the process tries to commit, *prepared* is the ballot that the process is safe to discard any ballots lower and incompatible with, and *leader* is the current leader. Each process uses an instance of federated voting for each ballot, and an eventual leader election module. The latter issues *new-leader* events, and eventually elects a well-behaved process as the leader. (Previous work [34] presented a probabilistic leader election module.)

Upon receiving a proposal request (at L. 11), a well-behaved process initializes its candidate ballot to the pair of the first round and its own proposal (at L. 12). 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. 14). 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. 15), it sets *prepared* to b (at L. 16). 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. 18). 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. 19), it decides the value of that ballot (at L. 20).

To ensure liveness, a well-behaved process triggers a timeout if no value is decided after a predefined time elapses in each round. The process then complains to the leader election module (at L. 22). When the leader election module issues a new leader (at L. 23), a well-behaved process updates its *leader* variable, and increments the *round* number (at L. 25). The leader itself then waits for a time Δ (at L. 27) which we will further explain below. The process also resets the timer with a doubled timeout for the next round (at L. 28). 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. 30); otherwise, it is updated to the new round number and the value of the *prepared* ballot (at L. 32). Then, the leader tries to prepare the *candidate* by aborting below and incompatible ballots similar to the steps above (at L. 34).

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 Figure 3. Consider that we have four processes, one of them 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 be prepared. The leader l_1 sends a commit for ballot b to one well-behaved process p_3. Then, p_3 echos commit for b. Then, the timeout for l_1 happens, and the next well-behaved leader l_2 comes up. Without the delay, l_2 may have not prepared b yet (although other well-behaved processes p_3 and p_4 prepared it). Therefore, the ballot b' that l_2 updates its candidate to (at L. 32) is not b, and may not be compatible with b. In order to prepare b', the leader l_2 tries to abort b (at L. 34) but b cannot be aborted: in order to abort b, a quorum of processes should echo it. However, the well-behaved process p_3 has already echoed commit, and if the Byzantine process l_1 remains silent, the remaining two well-behaved processes l_2 and p_4 are not a quorum, and cannot abort b. Therefore, l_2 cannot succeed, and the timeout is triggered. Further, if the next leader is the Byzantine process l_1 again, it can repeat the above scenario: it can abort b to 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. If the delay Δ is larger than the bounded communication delay after GST, it makes the leader l_2 observe the highest prepared ballot b, and adopt its value as the value of its candidate b'_2 (at L. 32). When it tries to commit b'_2, since it is compatible with b, it does not need abort it. Therefore, it can prepare and commit b'_2, and decide. 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.

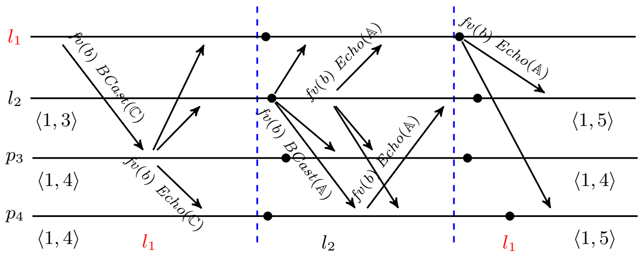

Figure 3 Last Minute Attack. $b = \langle 1, 4\rangle$. The *candidate* of well-behaved leader l_2 is $b' = \langle 2, 3\rangle$. The votes commit and abort are abbreviated as $\mathbb{C}$ and $\mathbb{A}$. The new leader events are triggered at the black dots at each process. Prepared ballots are shown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

▶ **Theorem 16.** *Quorum intersection and strong availability are sufficient to implement consensus.*

This theorem follows from three lemmas in the appendix [31] that prove that the protocol satisfies the specification of Byzantine consensus that we defined in Definition 11. An example execution of the protocol is described in the appendix [31].

7 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 [38, 33] presents federated Byzantine quorum systems (FBQS) [24, 25] where quorums are iteratively calculated from quorums slices. Stellar also presents a federated voting and consensus protocol. In comparison, the assumptions of the protocols presented in this paper are weaker, and their guarantees are stronger. 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. Further, abstract SCP [24] provides agreement only for intact processes. The intact set for an FBQS is a subset of processes that have strong availability. On the other hand, the consensus protocol in this paper provides agreement for all well-behaved processes. In FBQS, the intersections of quorums should have a process in the intact set; however, in HQS, they only need to have a well-behaved process. 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.

Personal Byzantine quorum systems (PBQS) [34] capture the quorum systems that FBQSs derive form slices, and propose a responsiveness consensus protocol [48, 1, 43, 3]. 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) [15, 16, 4] 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 [11, 10], and dynamic networks [41, 28]. It is interesting to see if their leader election modules can be generalized to the heterogeneous setting, and support responsiveness [48, 5] 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 [26, 19, 2, 21, 30, 8] 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.

8 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.

Acknowledgments

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 Ittai Abraham and Gilad Stern. Information theoretic hotstuff. *arXiv preprint arXiv:2009.12828*, 2020.

2 Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Consensus with byzantine failures and little system synchrony. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 147–155. IEEE, 2006.

3 Dan Alistarh, James Aspnes, Faith Ellen, Rati Gelashvili, and Leqi Zhu. Why extension-based proofs fail. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 986–996, 2019.

4 Orestis Alpos, Christian Cachin, and Luca Zanolini. 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 Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM (JACM)*, 41(1):122–152, 1994.

6 Mathieu Baudet, Avery Ching, Andrey Chursin, George Danezis, Francois Garillot, Zekun Li, Dahlia Malkhi, Oded Naor, Dmitri Perelman, and Alberto Sonnino. State machine replication in the libra blockchain. *The Libra Assn., Tech. Rep*, 7, 2019.

7 João Paulo Bezerra, Petr Kuznetsov, and Alice Koroleva. 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 Malte Borcherding. Levels of authentication in distributed agreement. In *International Workshop on Distributed Algorithms*, pages 40–55. Springer, 1996.

9 Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

10 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. 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 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making byzantine consensus live. *Distributed Computing*, 35(6):503–532, 2022.

12 Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains*. PhD thesis, University of Guelph, 2016.

13 Ethan Buchman, Rachid Guerraoui, Jovan Komatovic, Zarko Milosevic, Dragos-Adrian Seredinschi, and Josef Widder. 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 Christian Cachin, Giuliano Losa, and Luca Zanolini. Quorum systems in permissionless network. *arXiv preprint arXiv:2211.05630*, 2022.

15 Christian Cachin and Björn Tackmann. Asymmetric distributed trust. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

16 Christian Cachin and Luca Zanolini. From symmetric to asymmetric asynchronous byzantine consensus. *arXiv preprint arXiv:2005.08795*, 2020.

17 Harold Carr, Christa Jenkins, Mark Moir, Victor Cacciari Miraldo, and Lisandra Silva. 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 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

19 Carole Delporte-Gallet, Hugues Fauconnier, Rachid Guerraoui, Vassos Hadzilacos, Petr Kouznetsov, and Sam Toueg. 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 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

21 Michael J Fischer, Nancy A Lynch, and Michael Merritt. Easy impossibility proofs for distributed consensus problems. *Distributed Computing*, 1(1):26–39, 1986.

22 Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

23 Juan Garay and Aggelos Kiayias. Sok: A consensus taxonomy in the blockchain era. In *Cryptographers' track at the RSA conference*, pages 284–318. Springer, 2020.

24 Álvaro García-Pérez and Alexey Gotsman. Federated byzantine quorum systems. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

25 Álvaro García-Pérez and Maria A Schett. Deconstructing stellar consensus (extended version). *arXiv preprint arXiv:1911.05145*, 2019.

26 Guy Goren, Yoram Moses, and Alexander Spiegelman. Probabilistic indistinguishability and the quality of validity in byzantine agreement. *arXiv preprint arXiv:2011.04719*, 2020.

27 Martin Hirt, Ard Kastrati, and Chen-Da Liu-Zhang. Multi-threshold asynchronous reliable broadcast and consensus. *Cryptology ePrint Archive*, 2020.

28 Rebecca Ingram, Patrick Shields, Jennifer E Walter, and Jennifer L Welch. An asynchronous leader election algorithm for dynamic networks. In *2009 IEEE International Symposium on Parallel & Distributed Processing*, pages 1–12. IEEE, 2009.

29 Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19:104–125, 2006.

30 Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, 1982.

31 Xiao Li, Eric Chan, and Mohsen Lesani. Quorum subsumption for heterogeneous quorum systems. technical report. In *International Symposium on Distributed Computing (DISC 2023)*, 2023.

32 Xiao Li and Mohsen Lesani. Open heterogeneous quorum systems, 2023. `arXiv:2304.02156`.

33 Marta Lokhava, Giuliano Losa, David Mazières, Graydon Hoare, Nicolas Barry, Eli Gafni, Jonathan Jove, Rafał Malinowsky, and Jed McCaleb. Fast and secure global payments with stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 80–96, 2019.

34 Giuliano Losa, Eli Gafni, and David Mazières. Stellar consensus by instantiation. In *33rd International Symposium on Distributed Computing (DISC 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

35 Ethan MacBrough. Cobalt: Bft governance in open networks. *arXiv preprint arXiv:1802.07240*, 2018.

36 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1041–1053, 2019.

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

38 David Mazieres. The stellar consensus protocol: A federated model for internet-level consensus. *Stellar Development Foundation*, 32:1–45, 2015.

39 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

40 Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1686–1699, 2021.

41 Achour Mostefaoui, Michel Raynal, Corentin Travers, Stacy Patterson, Divyakant Agrawal, and Amr EL Abbadi. From static distributed systems to dynamic systems. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 109–118. IEEE, 2005.

42 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *White paper*, 2008.

43 Rafael Pass and Elaine Shi. 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 David Schwartz, Noah Youngs, and Arthur Britto. The ripple protocol consensus algorithm. *Ripple Labs Inc White Paper*, 5(8):151, 2014.

45 Isaac Sheff, Xinwen Wang, Robbert van Renesse, and Andrew C Myers. Heterogeneous paxos. In *OPODIS: International Conference on Principles of Distributed Systems*, number 2020 in OPODIS, 2021.

46 Isaac C Sheff, Robbert van Renesse, and Andrew C Myers. Distributed protocols and heterogeneous trust: Technical report. *arXiv preprint arXiv:1412.3136*, 2014.

47 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

48 Maofan Yin, Dahlia Malkhi, Michael K Reiter, Guy Golan Gueta, and Ittai Abraham. Hotstuff: BFT consensus with linearity and responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

Appendix

Contents

9 Byzantine Reliable Broadcast

9.1 Proofs

▶ **Theorem 17.** *Quorum intersection and strong availability are sufficient to implement Byzantine reliable broadcast.*

This theorem follows from the following five lemmas. In the following lemmas, we consider a quorum system with quorum intersection and strong availability for P.

▶ **Lemma 18.** *The BRB protocol guarantees consistency.*

Proof. A well-behaved process only delivers a message when it receives a quorum of consistent ready messages. If p_1 delivers m_1 with q_1, and p_2 delivers m_2 with q_2, by quorum intersection, there is well-behaved process p in $q_1 \cap q_2$. The process p sends ready messages with only one value. Thus, $m_1 = m_2$. ◀

▶ **Lemma 19.** *The BRB protocol guarantees validity for P.*

Proof. Consider a well-behaved sender that broadcasts a message m. We show that every process in P eventually delivers m. By availability, every process $p \in P$ has a complete quorum q. Consider a process $p' \in q$. By quorum subsumption, p' has a quorum $q' \subseteq q$. By availability, all members of q (including q') are well-behaved. Thus, when they receive m from the sender, they all echo it to their followers. The processes in q' have p' as a follower. Thus, p' receives consistent echo messages for m from one of its quorums q'. Thus, p' sends out ready messages for m to its followers. Thus, all processes in q send out ready messages for m to their followers. The processes in q have p as a follower. Therefore, p receives a quorum of consistent ready messages for m from one of its quorums q, and delivers m. ◀

▶ **Lemma 20.** *The BRB protocol guarantees totality for P.*

Proof. We assume that a well-behaved process p has delivered m, and show that every well-behaved process $p' \in P$ delivers m. We first show that every well-behaved process p'' sends ready messages for m to its followers.

The well-behaved process p delivers m only when it receives a quorum q of ready messages for m. Consider a quorum q''_i of p''. By quorum intersection, q and q''_i intersect in at least a well-behaved process p_i. Since p_i is in q''_i, then p_i has p'' as a follower. Let I be the union of the processes p_i for all quorums q''_i of p''. By construction, the set of processes I are a subset of q, a blocking set of p'', and have p'' as a follower. All well-behaved processes in q send ready messages to their followers. Thus, the set of processes I send ready messages to p''. Thus, p'' receives a blocking set of ready messages, and sends ready messages to its followers.

Thus, every well-behaved process p'' sends ready messages to their followers. By weak availability, $p' \in P$ has a quorum q' with all well-behaved members. Thus, the process p' will receive ready messages from q', and delivers m. ◀

▶ **Lemma 21.** *The BRB protocol guarantees no duplication.*

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

▶ **Lemma 22.** *The BRB protocol guarantees integrity.*

Proof. We assume that a well-behaved process p delivers a message m from a well-behaved sender process p', and prove that the process p' has broadcast m. We first prove that a well-behaved process has received echo messages from at least one of its quorums.

Since the well-behaved process p delivered m, it has received a quorum q_p of ready messages. By the strong availability assumption, P is non-empty; therefore, there exists a process with a quorum q that is well-behaved and quorum subsuming. By quorum intersection, q_p and q intersect at a well-behaved process p_i. The process p_i sent a ready message (that p received). The well-behaved process p_i sends a ready message only if it either (1) receives echo messages from a quorum of p_i, or (2) receives ready messages from a p_i-blocking set B. The first case is immediately the conclusion with the process p_i. Let us consider the second case. By quorum subsumption, since p_i is in q, p_i has a quorum q_i that is a subset of q. Since B is a blocking set of p_i, B intersects with q_i. Therefore, B intersects with q. The processes in B sent ready messages. Thus, there are processes in q that sent ready messages. Let p_f be the first process in q that sent a ready message. All members of q are well-behaved. A well-behaved process p_f sends a ready message only if it either (1) receives echo messages from a quorum of p_f, or (2) receives ready messages from a p_f-blocking set B'. The first case is immediately the conclusion with the process p_f. Let us consider the second case. Applying the same reasoning as for p_i to p_f derives that B' intersects with q. Thus, p_f has received a ready message from a process in q. Therefore, p_f is not the first process in q to send a ready message. This is a contradiction with the definition of p_f above.

We now use the fact that a well-behaved process has received echo messages from at least one of its quorums. 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 has sent an echo message. A well-behaved process sends an echo message only after receiving it from the sender p'. Since the sender p' is well-behaved, and p_w has received the message m from it, by the integrity of point-to-point links, p' has previously sent m to p_w. Since p' is well-behaved, it has sent m to every process. ◀

10 Byzantine Consensus

10.1 Proofs

▶ **Theorem 23.** *Quorum intersection and strong availability are sufficient to implement consensus.*

This theorem follows from the following three lemmas where we consider a quorum system with quorum intersection and strong availability for P.

▶ **Lemma 24** (Agreement). *The consensus protocol guarantees agreement.*

Proof. We prove agreement by contradiction. Assume that there are two well-behaved processes that decide two different values. A process decides a value at L. 20 after delivering a ballot with that value at L. 19. Let process p deliver v in ballot b, and let process p' deliver v' in ballot b'. Since ballots are totally ordered, without loss of generality, we assume that $b \lnsim b'$. When the process p' delivers v' at L. 19, it checks that b' has been prepared. A well-behaved process prepares a new ballot b' at L. 16 only after finding all the ballots below and incompatible with b' aborted at L. 15. Since $b \lnsim b'$, the process p' has delivered the $\mathbb{A}$ message for b before preparing b'. However, since process p decides v with b at L. 20, it has delivered $\mathbb{C}$ message for b at L. 19. Thus, two well-behaved processes deliver conflicting values $\mathbb{A}$ and $\mathbb{C}$ for ballot b, which violates the consistency property of federated voting, a contradiction. ◀

▶ **Lemma 25** (Validity). *The consensus protocol guarantees validity.*

Proof. We assume that all processes are correct, and that a process has decided a value v. We show that v is proposed by some process. When a well-behaved process decides a value v at L. 20, v was delivered in a $\mathbb{C}$ message for a ballot b at L. 19. Since all processes are well-behaved including the leader, by the integrity property of federated voting, the leader has broadcast $\mathbb{C}$ for the ballot b at L. 18. The ballot b is the leader's *candidate* by the condition at L. 17.

We show that the value of every *candidate* ballot can be tracked back to the proposal of a process. The value for the *candidate* ballot is set to either its initial proposal at L. 12 and L. 30, or adopted from the value of *prepared* at L. 32. We consider the two cases in turn. For the first case, the value v is immediately the leader's proposal at L. 11. For the second case, v is a value prepared before by the leader. A well-behaved process updates its *prepared* value to b at L. 16 when it delivers $\mathbb{A}$ for all $b' \lnsim b$ at L. 15. By the integrity of federated voting, and the assumption that all the processes are well-behaved, leaders have broadcast to abort all ballots b'. Leaders abort ballots below and incompatible with their *candidate* at L. 14 and L. 34. By the batched network semantics assumption, all the messages sent by a process in one event to another process are delivered and processed together. Therefore, the ballot b has been the *candidate* ballot of a previous leader at L. 14 or L. 34. With the same reasoning as above, the *candidate* value can be traced back to either a proposal, or the *candidate* value of a yet previous leader. Since the number of leaders is finite, by a well-founded induction, the *candidate* value is tracked back to the proposal of a process. ◀

The protocol guarantees termination: all strongly available processes eventually decide a value. The high-level idea is that by eventual election of a well-behaved leader, and the totality property of federated voting, a well-behaved leader eventually adopts the value of the highest prepared ballot in the system as its candidate Then, that leader can proceed to prepare and commit its candidate.

▶ **Lemma 26** (Termination). *The consensus protocol guarantees termination for P.*

Proof. The proof will refer to the notions of locked for federated voting. We first visit these notions, and then describe the proof of termination.

We saw in the explanation of the protocol that there are executions that Byzantine processes can lock a federated voting instance for a certain value (such as commit). Let us remember that example. Consider that we have 4 processes, one of them is Byzantine, and any set of 3 processes is a quorum. Let the Byzantine process send a message m to one well-behaved process. That well-behaved process echos m. If the Byzantine process remains silent, the remaining 2 well-behaved processes are not a quorum, and even if both echo another message m', they cannot make the federated voting deliver m'. This execution is shown in the first two rounds of messages in Figure 3. In this execution, the federated voting instance of b is locked for $m = \mathbb{C}$ by l_1 and p_3, and although l_2 and p_4 try to make it deliver $m' = \mathbb{A}$, it does not. However, if the Byzantine process echos m' as well, then they form a quorum, and can make m' be delivered. Well-behaved processes alone cannot make the instance deliver a different value, but well-behaved and Byzantine processes together can. We call these states of the federated voting abstraction soft-locked.

We note that there are also states of a federated voting instance where even if well-behaved and Byzantine processes work together, they cannot make the instance deliver a different value. We call them hard-locked. As above, consider an execution with 4 processes, and quorums of size 3. Let a Byzantine process send m to two well-behaved processes, and make them echo m. By the protocol, in order to deliver a message, a process needs to receive at least three ready messages. Further, in order to send a ready message, a process needs to receive at least either three echo messages or two ready messages. Therefore, to deliver any other message m', at least 3 processes should echo m'. Since 2 well-behaved processes out of 4 processes already echoed m, and well-behaved processes echo only once, there are not 3 processes to echo m', and make it delivered. The federated voting abstraction is hard-locked for m.

We now consider the proof of termination. By the eventual leader election, there will eventually be a well-behaved leader with a long enough timeout. Let b be the *candidate* ballot of the leader. The leader tries to abort all ballots lower and incompatible with b at L. 14 and L. 34. We consider three cases based on whether there is a ballot $b' \lesssim\!\!\!\!\not\,\, b$ that is locked for commit.

Case 1: There is no such ballot b'. The leader can abort all ballots below and incompatible with b at L. 15, and broadcast commit for b at L. 18. By the validity of federated voting, all processes in P eventually deliver commit for b at L. 19, and decide its value at L. 20.

Case 2: There is such a ballot b' that is soft-locked for commit. In order to prepare b, the leader tries to abort b'. If the Byzantine processes cooperate, it can abort b' and prepare b at L. 15, and broadcast commit for b at L. 18. However, if the Byzantine processes do not cooperate, the leader cannot abort b'. Therefore, the leader does not succeed, and the timeout is triggered. Eventually, a well-behaved process l will be the leader. Since the delay Δ is larger than the bounded communication delay after GST, the leader l observes the highest prepared ballot b at L. 15, and adopts its value as the value of its *candidate* b' at L. 17. We note that when it tries to prepare b', since b' is compatible with b, it does not need to abort b. Therefore, it can prepare and broadcast commit for b' at L. 18. By the validity of federated voting, all processes in P eventually deliver commit for b' at L. 19, and decide its value at L. 20.

Case 3: There is such a ballot b' that is hard-locked for commit. Ballot b' cannot be aborted. Therefore, no well-behaved process can prepare a ballot with a value incompatible with b'. The current leader will eventually timeout. Further, all ballots $b'' \lnsim b'$ should have been aborted since processes accept commit for b' only if b' is *prepared* at L. 19. By the totality of federated voting, eventually a well-behaved leader that has delivered abort for all ballots b'' will be elected. Therefore, that leader finds b' prepared at L. 15-16, and adopts its value as the value of its *candidate* ballot b''_2 at L. 32. Thus, b''_2 is compatible with b'. As we saw above, no ballot incompatible with b' can be prepared. Thus, no ballot incompatible with b''_2 can be prepared. Thus, the leader can successfully prepare b''_2 at L. 15, and broadcast commit for b''_2 at L. 18-17. By the validity of federated voting, all processes in P eventually deliver commit for b''_2 at L. 19, and decide its value at L. 20. ◀

10.2 Example Execution

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 $\langle 0, \bot \rangle$ 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. 11, all the well-behaved process update their *candidate* with their own proposal value at L. 12. Process 1 aborts all the ballots below and incompatible with its *candidate* through reliable broadcast: it $\mathit{BCast}\mathbb{A}$ for instances $\langle 0, \bot \rangle$, $\langle 1, 1 \rangle$ and $\langle 1, 2 \rangle$ at L. 14. Since the network is partially synchronized, we let the process 4 be partitioned from the rest of processes temporarily. Byzantine process 2 only send $\mathit{Echo}\mathbb{A}$ messages for ballot $\langle 0, \bot \rangle$ and $\langle 1, 1 \rangle$. Since process 1 has the individual minimal quorum $\{1, 2, 3\}$, it send $\mathit{Ready}\mathbb{A}$ messages for the aforementioned two ballots. Process 3 sends $\mathit{Ready}\mathbb{A}$ 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 $\langle 0, \bot \rangle$ and $\langle 1, 1 \rangle$ through BRB at L. 15 and update its *prepared* variable as $\langle 1, 2 \rangle$ at L. 16. Since the leader process 1's $\mathit{prepared} \neq \mathit{candidate}$, 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. 32; process 3 and 4 has not prepared any ballot and update their candidate to their original proposal at L. 30. 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. 19. 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. 15 and update their *prepared* ballot to $\langle 1, 2 \rangle$. No value is decided and the timer trigger again.

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 $\mathit{candidate} = \langle 3, 2 \rangle$ 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. 18. All the process 3 and 4 deliver the commit message at L. 19 and decide the value 2 in the committed ballot $\langle 3, 2 \rangle$ at L. 20.

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

Table 2 An execution for consensus protocol with leader switch