Reconfigurable Heterogeneous Quorum Systems

Xiao Li
University of California, Riverside, USA

Mohsen Lesani
University of California, Santa Cruz, USA

Abstract

In contrast to proof-of-work replication, Byzantine quorum systems maintain consistency across replicas with higher throughput, modest energy consumption, and deterministic liveness guarantees. If complemented with heterogeneous trust and open membership, they have the potential to serve as blockchains backbone. This paper presents a general model of heterogeneous quorum systems where each participant can declare its own quorums, and captures the consistency, availability and inclusion properties of these systems. In order to support open membership, it then presents reconfiguration protocols for heterogeneous quorum systems including joining and leaving of a process, and adding and removing of a quorum, and further, proves their correctness in the face of Byzantine attacks. The design of the protocols is informed by the trade-offs that the paper proves for the properties that reconfigurations can preserve. The paper further presents a graph characterization of heterogeneous quorum systems, and its application for reconfiguration optimization.

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

Keywords and phrases Quorum Systems, Reconfiguration, Heterogeneity

Digital Object Identifier 10.4230/LIPIcs.DISC.2024.37

1 Introduction

Banks have been traditionally closed; only established institutions could hold accounts and execute transactions. With regulations in place, this centralized model can preserve the integrity of transactions. However, it makes transactions across these institutions costly and slow; further, it keeps the power in the hands of a few. In pursuit of decentralization, Bitcoin [53] provided open membership: any node can join the Bitcoin network, and validate and process transactions. It maintains a consistent replication of an append-only ledger, called the blockchain, on a dynamic set of global hosts including potentially malicious ones. However, it suffers from a few drawbacks: low throughput, high energy consumption, and only probabilistic guarantees of commitment [40, 41].

Maintaining consistent replication in the presence of malicious processes has been the topic of Byzantine replicated systems for decades. PBFT [20] and its numerous following variants [63, 51, 65, 61, 8, 62] can maintain consistent replication when the network size is at least three times the size of potentially Byzantine coalitions, have higher throughput than Bitcoin, have modest energy consumption, give participants equal power, and provide deterministic liveness guarantees. Unfortunately, however, their quorums are uniform and their membership is closed. Their trust preferences, *i.e.*, the quorums of processes are fixed and homogeneous across the network. Further, their set of participants are fixed; thus, in contrast to proof-of-work replication that provides permissionless blockchains, classical Byzantine replication only provides permissioned blockchains.

Can the best of both worlds come together? Can we keep the consistency, throughput, modest energy consumption and equity of Byzantine replicated systems, and bring heterogeneous trust [24, 19, 4] and *open membership* to it? Openness challenges classical assumptions. With global information about the processes and their quorums, classical quorum systems

© Xiao Li, Mohsen Lesani;
licensed under Creative Commons License CC-BY 4.0
38th International Symposium on Distributed Computing (DISC 2024).
Editor: Dan Alistarh; Article No. 37; pp. 37:1–37:46
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

could be configured at the outset to satisfy consistency and availability properties. However, open quorum systems relinquish global information as processes specify their own quorums, and can further join, leave, and reconfigure their quorums. As the other processes may be unaware of these changes, consistency and availability may be violated after and even while these reconfigurations happen.

Projects such as Ripple [58] and Stellar [50] pioneered, and follow-up research [44, 43, 30, 12] moved towards this goal, and presented quorum systems where nodes can specify their own quorums, and can join and leave. In fact, the Stellar network has a high churn. In previous works, the consistency of the network is either assumed to be maintained by user preferences or a structured hierarchy of nodes, is provided only in divided clusters of processes, or can be temporarily violated and is periodically checked across the network. Reconfigurations can compromise the consistency or availability of the replicated system. The loss of consistency can be the antecedent to a fork and double-spending. An important open problem is *reconfiguration protocols for heterogeneous quorum systems with provable security guarantees*. The protocols are expected to avoid external central oracles, or downtime.

In this paper, we first present a *general model of heterogeneous quorum systems* where each process declares its individual set of quorums, and then formally capture the properties of these systems: consistency, availability and inclusion. We then consider the *reconfiguration* of heterogeneous quorum systems: joining and leaving of a process, and adding and removing of a quorum. To cater for the protocols such as broadcast and consensus that use the quorum system, the reconfiguration protocols are expected to preserve the above properties.

The safety of consensus naturally relies on the *consistency (or quorum intersection)* property: every pair of quorums intersect at a well-behaved process. Intuitively, if an operation communicates with a quorum, and a later operation communicates with another quorum, only a well-behaved process in their intersection can make the second aware of the first. A quorum system is *available* for a process if it has a well-behaved quorum for that process. Intuitively, the quorum system is responsive to that process through that quorum. The less known property is *quorum inclusion*. Roughly speaking, every quorum should include a quorum of each of its members. This property trivially holds for homogeneous quorum systems where every quorum is uniformly a quorum of all its members, but should be explicitly maintained for heterogeneous quorum systems. We show that quorum inclusion interestingly lets processes in the included quorum make local decisions while preserving properties of the including quorum. We precisely capture and illustrate these properties.

We then present *quorum graphs*, a graph characterization of heterogeneous quorum systems with the above properties. It is known that strongly connected components of a graph form a directed acyclic graph (DAG). We prove that a quorum graph has only one *sink component*, and preserving consistency reduces to preserving quorum intersections in this component. This fact has an important implication for optimization of reconfiguration protocols. Any change outside the sink component preserves consistency, and therefore, can avoid synchronization with other processes. Thus, we present a decentralized *sink discovery protocol* that can find whether a process is in the sink.

In addition to consistency, availability and inclusion, reconfiguration protocols are expected to preserve *policies*. Each process declares its own trust policy: it specifies the quorums that it trusts. In particular, it does not trust strict subsets of its individual quorums. Thus, a policy-preserving reconfiguration should not shrink any quorum. We present a *join protocol* that preserves all the above properties. We present *trade-offs* for the properties that the leave, remove and add reconfiguration protocols can preserve. We show that there is no *leave or remove protocol* that can preserve both the policies and availability. Thus, we present two protocols: a protocol that preserves policies, and another that preserves availability. Both preserve consistency and inclusion. Then, we show that there is no *add protocol* that can preserve both the policies and consistency. Therefore, since we never sacrifice consistency, we present a protocol that preserves all properties except the policies.

We observe that under reconfiguration, *quorum inclusion is critical* to preserve not only availability but also consistency. Sometimes, reconfigurations can only eventually reconstruct inclusion, but can preserve *weaker notions of inclusion* that are sufficient to preserve consistency and availability. We capture these notions, prove that they are preserved, and use them to prove that the other properties are preserved.

2 Quorum Systems

Processes. A quorum system is hosted on a set of processes $\mathcal{P}$. In each execution, $\mathcal{P}$ is partitioned into *Byzantine* $\mathcal{B}$ and *well-behaved* $\mathcal{W} = \mathcal{P} \setminus \mathcal{B}$ processes. Well-behaved processes follow the given protocols; however, Byzantine processes can deviate from the protocols arbitrarily. Furthermore, a well-behaved process does not know the set of well-behaved processes $\mathcal{W}$ or Byzantine processes $\mathcal{B}$. The active processes $\mathcal{A} \subseteq \mathcal{P}$ are the current members of the system. As we will see in Section 5, quorum systems can be reconfigured, and the active set can change: processes can join and the active set grows, and conversely, processes can leave, and the active set shrinks.

We consider partially synchronous networks [27], *i.e.*, if both the sender and receiver are well-behaved, the message will be eventually delivered within a bounded delay after an unknown GST (Global stabilization Time). Processes can exchange messages on authenticated point-to-point links.

Individual Quorums. Processes can have different trust assumptions: trust is a subjective matter, and therefore, heterogeneous. We capture a heterogeneous model of quorum systems where each process can specify its individual set of quorums.

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

Figure 1 Example Quorum System

An *individual quorum* q of a process p is a non-empty subset of processes in $\mathcal{P}$ that p trusts to collectively perform an operation. Every quorum of a process p naturally contains p itself. (However, this is not necessary for any theorem in this paper.) By the above definition, any superset of a quorum of p is also a quorum of p. Thus, the set of quorums of p is superset-closed and has minimal members. (Consider a set of sets $S = \{\overline{s}\}$. We say that S is superset-closed, if any superset s' of any member s of S is a member of S as well.) For example, let the minimal quorums of process 1 be the set $\{\{1,4\},\{1,3\}\}$. Then, the set $\{1,3,4\}$ is a quorum of 1 but is not a minimal quorum of 1. A process p doesn't need to keep any quorum other than its minimal quorums: any of its other quorums include extra processes that p can perform operations without. Thus, we consider only the *(individual) minimal quorums* of p. Any superset of such a quorum is a *quorum* for p. We denote a set of quorums as Q. We denote the union of a set of quorums Q as $\cup Q$.

Heterogeneous Quorum Systems. In a heterogeneous quorum system, the set of individual quorums can be different across processes.

▶ **Definition 1** (Quorum System). *A heterogeneous quorum system (HQS) $\mathcal{Q}$ maps each active process to a non-empty set of individual minimal quorums.*

The mapping models the fact that each process has only a local view of its own individual minimal quorums. Consider the running example in Figure 1. The minimal quorums of process 2 are $\{1, 2\}$, $\{2, 3\}$ and $\{2, 5\}$. Further, since the behavior of Byzantine processes can be arbitrary, we leave their individual quorums unspecified.

When obvious from the context, we say quorum systems to refer to heterogeneous quorum systems, and say quorums of p to concisely refer to the individual minimal quorums of p.

Quorums. Next, we consider quorums and their minimality across all processes of a quorum system. Consider a quorum system $\mathcal{Q}$. The set of (individual) quorums of $\mathcal{Q}$ is the set of quorums in the range of the map $\mathcal{Q}$. A quorum q is a *minimal quorum* of $\mathcal{Q}$ iff q is an individual minimal quorum of a process in $\mathcal{Q}$, and no proper subset of q is an individual minimal quorum of any process in $\mathcal{Q}$. (A minimal quorum is also called elementary [44].) We denote the set of minimal quorums of $\mathcal{Q}$ as $MQ(\mathcal{Q})$. In our running example in Figure 1, $MQ(\mathcal{Q}) = \{\{1, 2\}, \{2, 3\}, \{2, 5\}\}$. We note that although $\{1, 2, 4\}$ is a minimal quorum of 1, it is not a minimal quorum of $\mathcal{Q}$ since since 2 has the quorum $\{1, 2\}$ that is a strict subset of $\{1, 2, 4\}$.

▶ **Lemma 2.** *For all quorum systems $\mathcal{Q}$, every minimal quorum of $\mathcal{Q}$ is an individual minimal quorum of some process in $\mathcal{Q}$. Further, every quorum of $\mathcal{Q}$ is a superset of a minimal quorum of $\mathcal{Q}$.*

We note that a quorum that is not a strict superset of a minimal quorum is a minimal quorum itself.

3 Properties

The *consistency, availability and inclusion* properties are expected to be provided by a quorum system, and maintained by a reconfiguration protocol. In this section, we precisely define these notions. We adapt consistency and availability for HQS, and define the new notion of inclusion. We then consider a few variants of HQS. The conditions are parametric for a Byzantine attack, *i.e.*, the set of Byzantine processes $\mathcal{B}$ (or equivalently the set of well-behaved processes $\mathcal{W}$). Each condition can be directly lifted for a set of attacks $\{\overline{\mathcal{B}}\}$ by requiring the condition for each $\mathcal{B}$.

Consistency. A process stores and retrieves information from the quorum system by communicating with one of its quorums. Therefore, to ensure that each operation observes the previous one, the quorum system is expected to maintain an intersection for any pair of quorums at well-behaved processes. A set of quorums have quorum intersection at a set of well-behaved processes $P \subseteq \mathcal{W}$ iff every pair of them intersect in at least one process in P.

▶ **Definition 3** (Consistency, Quorum Intersection)**.** *A quorum system $\mathcal{Q}$ is consistent (i.e., has quorum intersection) at a set of well-behaved processes P iff the quorums of well-behaved processes have quorum intersection at P, i.e., $\forall p, p' \in \mathcal{W}.\ \forall q \in \mathcal{Q}(p),\ q' \in \mathcal{Q}(p').\ q \cap q' \cap P \neq \emptyset$.*

The set P is often implicitly the set of all well-behaved processes $\mathcal{W}$.

For example, in Figure 1, the quorum system $\mathcal{Q}$ is consistent since any two quorums have a well-behaved process (either 1 or 2) in their intersection. It is straightforward that every minimal quorum of a consistent quorum system contains a well-behaved process.

▶ **Lemma 4.** *In every quorum system, minimal quorums have quorum intersection iff individual minimal quorums have quorum intersection.*

Immediate from Lemma 2. This has an important implication for preservation of consistency.

▶ **Lemma 5.** *Every quorum system is consistent if its minimal quorums have quorum intersection.*

Straightforward from Definition 3 and Lemma 4.

Availability. To support progress for a process, the quorum system is expected to provide at least one responsive quorum for that process.

▶ **Definition 6** (Availability). *A quorum system is available for processes P at a set of well-behaved processes P' iff every process in P has at least a quorum that is a subset of P'.*

We say that a quorum system is available for P iff it is available for P at the set of active well-behaved processes. In our running example in Figure 1, the quorum system $\mathcal{Q}$ is available for $\{2, 3, 5\}$ since the processes 2 and 3 have the quorum $\{2, 3\}$, process 5 has the quorum $\{2, 5\}$, and the members of the quorums, 2, 3 and 5, are well-behaved. We note that $\mathcal{Q}$ is not available for 1 since its quorum intersects Byzantine processes $\mathcal{B} = \{4\}$.

We say that a quorum system is available *inside* P iff it is available for P at P. The set P has an interesting property that we will later use to maintain consistency. Consider a process p in P. If a set of processes P' can block availability for p, then P' intersects P. In our running example in Figure 1, the quorum system $\mathcal{Q}$ is available inside $P = \{2, 3, 5\}$. The set $P' = \{1, 3, 5\}$ intersects all quorums of process 2 and can block its availability. We observe that the two sets P and P' intersect.

Let's first see the notion of blocking set [44, 29] for quorums (rather than slices [50]).

▶ **Definition 7** (Blocking Set). *A set of processes P is a blocking set for a process p (or is p-blocking) iff P intersects every quorum of p.*

▶ **Lemma 8.** *In every quorum system that is available inside a set of processes P, every blocking set of every process in P intersects P.*

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

Quorum inclusion. Before defining the notion of quorum inclusion, let us start with an intuitive example of how inclusion of quorums can support their intersection. Consider a pair of quorums q_1 and q_2 that intersect at a well-behaved process p. Let a quorum q'_1 of p be included in q_1, and a quorum q'_2 of p be included in q_2. Consider that p wants to check whether it can leave without violating quorum intersection for q_1 and q_2. It is sufficient that p locally checks if there is at least one well-behaved process in the intersection of its own quorums q'_1 and q'_2.

Let us start with a simple example. The quorum system $\mathcal{Q}$ in Figure 1 is quorum including (for $\mathcal{W}$). For example, consider process $p = 2$, and the quorum $q = \{1, 2\}$ of p. The well-behaved processes p' of q are 1 and 2. Process 1 has the quorum $q' = \{1, 2, 4\}$ and its well-behaved subset is $\{1, 2\}$ that is included in q. Process 2 has quorum q that is trivially a subset of itself. Figure 2 illustrates the following definition of quorum inclusion.

▶ **Definition 9** (Quorum inclusion). *Consider a quorum system $\mathcal{Q}$, and a subset P of its well-behaved processes.*

A quorum q is quorum including for P iff for every process p in the intersection of q and P, there is a quorum q' of p such that well-behaved processes of q' are a subset of q, i.e., including$(q, P) := \forall p \in q \cap P.\ \exists q' \in \mathcal{Q}(p).\ q' \cap \mathcal{W} \subseteq q$.

A quorum system $\mathcal{Q}$ is quorum including for P iff every quorum of well-behaved processes of $\mathcal{Q}$ is quorum including for P, i.e., $\forall p \in \mathcal{W}.\ \forall q \in \mathcal{Q}(p).\ including(q, P)$.

The set P is often implicitly the set of all well-behaved processes $\mathcal{W}$.
Quorum inclusion was inspired by and weakens quorum sharing [44].

▶ **Definition 10** (Quorum sharing). *A quorum q has* quorum sharing *iff for every process p in q, there exists a quorum q' of p that is a subset of q. A quorum system has quorum sharing if all its quorums have quorum sharing, i.e., $\forall p, \forall q \in \mathcal{Q}(p).\ \forall p' \in q.\ \exists q' \in \mathcal{Q}(p').\ q' \subseteq q$.*

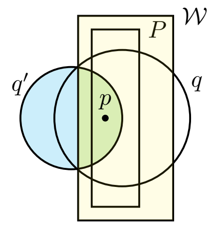


Figure 2 Quorum inclusion of q for P. Process p is a member of q that falls inside P, and q' is a quorum of p. Well-behaved processes of q' (shown as green) should be a subset of q.

Quorum sharing requires conditions on the Byzantine processes in q and q', and is too strong to maintain. We presented quorum inclusion that is weaker than quorum sharing. It requires a quorum q' only for well-behaved processes of q, and requires only the well-behaved subset of q' to be a subset of q. We will see in Section 6 that quorum inclusion is sufficient to support quorum intersection.

Outlived. As we will see in our reconfiguration protocols, quorum inclusion and quorum availability support quorum intersection. Thus, we tightly integrate these three properties in the notion of *outlived quorum systems.*

▶ **Definition 11** (Outlived). *A quorum system $\mathcal{Q}$ is outlived for a set of well-behaved processes $\mathcal{O}$ iff (1) $\mathcal{Q}$ is consistent at $\mathcal{O}$, (2) available inside $\mathcal{O}$, and (3) quorum including for $\mathcal{O}$.*

In an outlived quorum system, well-behaved processes enjoy safety (quorum intersection) and outlived processes enjoy liveness (availability of a quorum with inclusion). The safety and liveness properties of outlived processes outlive Byzantine attacks, hence the name. For example, our running quorum system in Figure 1 is outlived for $\{2, 3, 5\}$.

We call $\mathcal{O}$ an *outlived set* for $\mathcal{Q}$, and call a member of $\mathcal{O}$ an *outlived process.* We call a quorum system that is outlived for a set, an outlived quorum system. Similarly, we use the qualifier outlived for the properties (1)-(3) above. Quorum systems are initialized to be outlived, and the reconfiguration protocols preserve this property.

HQS Instances. We now describe a few instances of HQS, and their properties.

Dissemination quorum systems (DQS). A DQS [49] (and the cardinality-based quorum system as a a special case) declares a global set of quorums for all processes. Processes have the same set of individual minimal quorums. DQS further declares a set of possible Byzantine sets. A DQS is outlived for all well-behaved processes $\mathcal{W}$. It is consistent at $\mathcal{W}$ since the intersection of no pair of quorums falls completely in a Byzantine set. It is available for $\mathcal{W}$ since there is at least one quorum that does not intersect with any Byzantine set. It is quorum including for $\mathcal{W}$: since the quorums are global, all the well-behaved members of a quorum q recognize q as their own quorum. However, in general, an HQS may be outlived for only a subset of well-behaved processes.

Personal Byzantine quorum systems (PBQS). A PBQS [44] is an HQS that requires quorum sharing, and further quorum intersection and availability for subsets of processes called clusters. A cluster is an outlived HQS.

Federated (Byzantine) quorum systems (FBQS). An FBQS [50, 29] 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, the resulting HQS enjoys quorum sharing [44]. 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, there is a slice s of process p' in q. Since Byzantine processes don't lie about their slices, processes receive the same set of slices from a given process. If p' starts from s, it can gather the same slices for the processes s as p does, and can assemble a quorum q' that grows no larger than q. Therefore, q is a superset of a quorum q'. However, if Byzantine processes lie about their slices, quorum sharing may not hold.

4 Graph Characterization

We now define a graph characterization of heterogeneous quorum systems. We show that the graphs for quorum systems with certain properties have a single sink component that contains all the well-behaved processes in minimal quorums; therefore, by Lemma 5, preserving consistency reduces to preserving quorum intersections in that component.

Quorum graph. The quorum graph of a quorum system $\mathcal{Q}$ is a directed graph $G = (\mathcal{P}, E)$, where vertices are the processes, and there is an edge from p to p' if p' is a member of an individual minimal quorum of p, *i.e.*, $(p, p') \in E$ iff $\exists q \in \mathcal{Q}(p).\ p' \in q$. Intuitively, the edge (p, p') represents the fact that p directly consults with p'. For example, Figure 3 shows a quorum system and its graph representation. We refer to a quorum system and its graph characterization interchangeably.

We now prove a few properties for quorum systems with consistency and quorum sharing. (Quorum systems with these properties enable optimizations for reconfiguration; however, the protocols in the next sections don't require quorum sharing.)

▶ **Lemma 12.** *A quorum is a minimal quorum iff it is an individual minimal quorum for all its well-behaved members.*

Proof. We first show the only-if direction. Consider a minimal quorum q. By the quorum sharing property, each well-behaved process in q has an individual minimal quorum q' such that $q' \subseteq q$. Since q is a minimal quorum, $q' = q$. The proof of the if direction is by contradiction. Assume the if condition: q is an individual minimal quorum for all its well-behaved members. However, q is not a minimal quorum. Since q is an individual minimal quorum but not a minimal quorum, by Lemma 2, there is a minimal quorum q' such that $q' \subsetneq q$. Let p be a well-behaved process in q' (and therefore, q). By the if condition, q is an individual minimal quorum of p. By the only if direction, q' is an individual minimal quorum of p. However, these two facts and $q' \subsetneq q$ contradict the minimality assumption for q. ◀

We remember that a subset of vertices that are pair-wise connected are a clique.

▶ **Lemma 13.** *Well-behaved processes in a minimal quorum are a clique.*

This is immediate from Lemma 12. For example, in Figure 3, the minimal quorums are $MQ(\mathcal{Q}) = \{\{1, 2\}, \{1, 3, 5\}\}$, and their well-behaved processes $\{1, 2\}$ and $\{1, 3\}$ are cliques.

▶ **Lemma 14.** *Every well-behaved process is adjacent to all processes of a minimal quorum.*

Proof. By Definition 1, a well-behaved process p has at least one quorum q. Process p has an edge to every member of q. By Lemma 2, q is a superset of a minimal quorum q'. Therefore, p has an edge to every member of q'. ◀

In Figure 3, process 4 is adjacent to all processes of $\{1, 2\}$.

$\mathcal{P} = \{1, 2, 3, 4, 5, 6\}, \mathcal{B} = \{5\}$,
$\mathcal{Q}(1) = \{\{1, 2\}, \{1, 3, 5\}\}$,
$\mathcal{Q}(2) = \{\{1, 2\}\}$,
$\mathcal{Q}(3) = \{\{1, 3, 5\}\}$,
$\mathcal{Q}(4) = \{\{1, 2, 4\}\}$,
$\mathcal{Q}(5) = \{\{1, 3, 5\}\}$,
$\mathcal{Q}(6) = \{\{1, 2, 6\}\}$
$MQ(\mathcal{Q}) = \{\{1, 2\}, \{1, 3, 5\}\}$

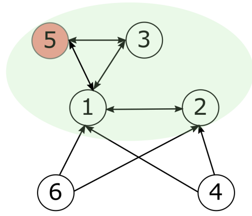


Figure 3 Quorum Graph Example

▶ **Lemma 15.** *Well-behaved processes of minimal quorums induce a strongly connected graph.*

Proof. Consider a pair of minimal quorums q_1 and q_2, and two well-behaved processes $p_1 \in q_1$ and $p_2 \in q_2$. The consistency property states that there is at least a well-behaved process p in the intersection of q_1 and q_2. By Lemma 13, the following edges are in the quorum graph: (p_1, p), (p, p_2), (p_2, p) and (p, p_1). Therefore, p_1 and p_2 are strongly connected. ◀

In Figure 3, the processes $\{1, 2, 3\}$ are strongly connected.

We remember that the condensation of a graph is the graph resulted from contracting each of its strongly connected components to a single vertex. A condensation graph is a directed acyclic graph (DAG). DAGs have sink and source vertices. A component of the graph that is contracted to a sink vertex in the condensed graph is called a sink component.

▶ **Lemma 16.** *All well-behaved processes in minimal quorums are in a sink component.*

Proof. By Lemma 15, the well-behaved processes of the minimal quorums are a strongly connected subgraph. Therefore, they fall in a component C. By Lemma 14, there are edges from the processes of every component to C. Therefore, C must be a sink component. ◀

For example, in Figure 3, processes $\{1, 2, 3\}$ (shaded in green) are in the sink.

▶ **Lemma 17.** *There exists a minimal quorum in every sink component.*

Proof. Consider a sink component S. By Lemma 14, there are edges from S to all processes of a minimal quorum q. This quorum q should be inside S. Otherwise, the fact that there are edges from S to q contradicts the assumption that S is a sink component. ◀

▶ **Lemma 18.** *Every quorum graph has only one sink component.*

Proof. The proof is by contradiction. If there are two sinks, by Lemma 17, each contains a minimal quorum. By the quorum intersection property, the two minimal quorums have an intersection; thus, the two sinks components intersect. However, components are disjoint. ◀

▶ **Theorem 19.** *All well-behaved processes of the minimal quorums are in the sink component.*

This is straightforward from Lemma 16 and Lemma 18. For example, in Figure 3, the well-behaved processes $\{1, 2, 3\}$ of the minimal quorums $\{1, 2\}$ and $\{1, 3, 5\}$ are in the sink.

Consider a reconfiguration from a quorum system $\mathcal{Q}$ to another $\mathcal{Q}'$, and a well-behaved process p. A *Leave* operation by p removes p from the set of active processes $\mathcal{A}$ *i.e.*, $p \notin \mathsf{dom}(\mathcal{Q}')$. Let q be an individual minimal quorum of p, *i.e.*, $q \in \mathcal{Q}(p)$. A *Remove*(q) operation by p removes q from the individual minimal quorum of p, *i.e.*, $q \notin \mathcal{Q}'(p)$.

▶ **Lemma 20.** *Any leave or remove operation by a process outside the sink component of the quorum graph preserves consistency.*

Proof. By Lemma 5, it is sufficient to prove that quorum intersection is preserved for minimal quorums. By Theorem 19, the well-behaved intersections of minimal quorums fall in the sink component. Therefore, any leave or remove operation outside of the sink component preserves their quorum intersection. ◀

Inspired by this result, our leave and remove protocols will avoid coordination when they are applied to a process that is outside of the sink component. (We will present a sink discovery protocol in the appendix Section 15).

5 Reconfiguration and Trade-offs

In this section, we consider reconfigurations, how they can endanger the properties of a quorum system, and trade-off theorems for the properties that reconfiguration protocols can preserve. These trade-offs inform the design of our protocols in the next sections.

A process can request to *Join* or *Leave* the quorum system. It can further request to *Add* or *Remove* a quorum. However, a reconfiguration operation should not affect the safety and liveness of the quorum system.

Reconfiguration Attacks. Let $\mathcal{P} = \{1, 2, 3, 4\}$ where the Byzantine set is $\mathcal{B} = \{4\}$. Let the quorums of process 1 be $\mathcal{Q}(1) = \{\{1, 2, 4\}\}$. Similarly, let $\mathcal{Q}(2) = \{\{1, 2\}, \{2, 3\}\}$ and $\mathcal{Q}(3) = \{\{2, 3\}\}$. This quorum system enjoys quorum intersection for well-behaved processes since all pairs of quorums intersect at a well-behaved process. Let process 2 locally add a quorum $q_1 = \{2, 4\}$ its set of quorums $\mathcal{Q}(2)$. The quorum q_1 intersects all the existing quorums at the well-behaved process 2. Similarly, let process 3 locally add a quorum $q_2 = \{1, 3\}$ into its set of quorums $\mathcal{Q}(3)$. The quorum q_2 intersects all the existing quorums at the well-behaved processes 1 or 3. Both reconfiguration requests seem safe, and if they are requested concurrently, they may be both permitted. However, the two new quorums do not intersect. An attacker can issue a transaction to spend some credit at process 2 with q_1, and another transaction to spend the same credit at process 3 with q_2. That leads to a double-spending and a fork. Even if processes 2 and 3 send their updated quorums to other processes, the attack can be successful if the time to send and receive updates is longer than the time to process a transaction. Similarly, a leave operation can lead to double-spending. In our example, if process 2 leaves the system, quorum intersection is lost. The reconfiguration protocols should preserve quorum intersection.

Trade-offs. We first formalize a few notions to state the trade-offs.

Reconfigurations. A reconfiguration changes a quorum system to another. We remember that a quorum system is a mapping from active well-behaved processes $\mathcal{A} \cap \mathcal{W}$ to their quorums. Consider a reconfiguration by a well-behaved process p that updates $\mathcal{Q}$ to $\mathcal{Q}'$. We consider four reconfiguration operations. The reconfiguration applies a *Join* operation by p iff $p \notin \mathsf{dom}(\mathcal{Q})$ and $p \in \mathsf{dom}(\mathcal{Q}')$ (*i.e.*, p is added to the active set $\mathcal{A}$). It applies an *Add*(q) operation by p iff $q \in \mathcal{Q}'(p)$. It applies a *Leave* operation by p iff $p \in \mathsf{dom}(\mathcal{Q})$ and $p \notin \mathsf{dom}(\mathcal{Q}')$ (*i.e.*, p is removed from the active set $\mathcal{A}$). It applies a *Remove*(q) operation by p where $q \in \mathcal{Q}(p)$ (*i.e.*, q is an individual minimal quorum of p) iff $q \notin \mathcal{Q}'(p)$.

Terminating. A reconfiguration protocol is terminating iff every operation by a well-behaved process eventually completes.

Each process declares its trust policy as its individual minimal quorums. A quorum should appear in the individual minimal quorums of a process only if that process has

$\mathcal{Q}_1(1) = \_$	$\mathcal{Q}_2(1) = \_$
$\mathcal{Q}_1(2) = \{\{2, 3\}, \{1, 2, 4\}\}$	$\mathcal{Q}_2(2) = \{\{2, 3\}, \{1, 2\}\}$
$\mathcal{Q}_1(3) = \{\{2, 3\}, \{1, 3, 4\}\}$	$\mathcal{Q}_2(3) = \{\{2, 3\}, \{3, 4\}\}$
$\mathcal{Q}_1(4) = \{\{1, 3, 4\}\}$	$\mathcal{Q}_2(4) = \{\{1, 3, 4\}\}$

Figure 4 Example Quorum Systems for Trade-offs

explicitly declared it as its quorum, during either the initialization or an add reconfiguration.

Policy-preservation. A *Leave* or *Remove* operation is policy-preserving iff it only removes individual minimal quorums. A *Join* operation is policy-preserving iff it does not change existing individual minimal quorums. An *Add*(q) operation by a process p is policy-preserving iff it only adds q to individual minimal quorums of p.

Consistency-preservation. A reconfiguration is consistency-preserving iff it transforms a consistent quorum system to only a consistent one.

Availability-preservation. A reconfiguration is availability-preserving iff it only affects the availability of a process that is requesting *Leave*, or requesting *Remove* for its last quorum.

▶ **Theorem 21.** *There is no* Leave *or* Remove *reconfiguration protocol that is policy-preserving, availability-preserving and terminating.*

Proof. The proof is by contradiction. We consider the Leave and Remove protocols in turn.

The Leave protocol: Consider the quorum system $\mathcal{Q}_1$ in Figure 4. Process 1 is Byzantine. $\mathcal{Q}_1(2) = \{\{2, 3\}\}$. (We will later reuse this example for the Remove protocol after adding the quorum $\{1, 2, 4\}$ for process 2, as the figure shows in color.) $\mathcal{Q}_1$ is available for $\{2, 3\}$. Process 2 requests to leave. Since the protocol is terminating, 2 eventually leaves, and the quorum system is updated to $\mathcal{Q}'_1$. The quorum $\{2, 3\}$ makes 3 available in $\mathcal{Q}_1$ but not $\mathcal{Q}'_1$. If the protocol leaves the quorum $\{2, 3\}$ unchanged, then it includes the inactive process 2. The other quorum $\{1, 3, 4\}$ of 3 includes the Byzantine process 1. Thus, $\mathcal{Q}'_1$ does not preserve availability for 3. If the protocol removes 2 from the quorum $\{2, 3\}$, then $\mathcal{Q}'_1$ preserves availability for 3 but does not preserve policies.

The Remove protocol: We reuse the example above with a small change: $\mathcal{Q}_1(2) = \{\{2, 3\}, \{1, 2, 4\}\}$. Let process 2 remove quorum $\{2, 3\}$ and result in quorum system $\mathcal{Q}'_1$. Now, $\mathcal{Q}'_1$ loses availability for 2 unless process 2 removes 1 from its quorum $\{1, 2, 4\}$. However, that violates policies. ◀

▶ **Theorem 22.** *There is no* Add *reconfiguration protocol that is policy-preserving, consistency-preserving, and terminating.*

Proof. Consider the quorum system $\mathcal{Q}_2$ in Figure 4. Process 1 is Byzantine. Process 2 requests to add a new quorum $\{1, 2\}$ that is shown in color. Since the protocol is terminating, it will eventually add $\{1, 2\}$ to the quorums of 2, and result in the updated quorum system $\mathcal{Q}'_2$. The quorum system $\mathcal{Q}_2$ is consistent for $\{2, 3, 4\}$. However, in $\mathcal{Q}'_2$, the quorum $\{1, 2\}$ of process 2, and the quorum $\{1, 3, 4\}$ of the process 4 intersect at only the Byzantine process 1. Therefore, to preserve consistency, there are two cases. In the first case, $\{1, 3, 4\}$ is removed from the quorums of 4. Then, the well-behaved active process 4 has no quorums which violates the definition of heterogeneous quorum systems. In the second case, 2 is added to $\{1, 3, 4\}$. However, this violates policies for 4. ◀

6 Leave and Remove

In the light of these trade-offs, we next consider reconfiguration protocols. The protocols reconfigure an outlived quorum system into another. They assume that the given quorum system is outlived, *i.e.*, it has an outlived set of processes $\mathcal{O}$. In particular, they only require quorum inclusion (and not quorum sharing) inside $\mathcal{O}$. Let's now consider Leave and Remove protocols. (The Join protocol is straightforward and presented in Section 11.) The client can issue a *Leave* request to leave the quorum system, and in return receives either a *LeaveComplete* or *LeaveFail* response. It can also issue the *Remove*(q) request to remove its quorum q, and in return receives either a *RemoveComplete* or *RemoveFail* response. Based on the trade-offs that we saw in Theorem 21, we present the availability-preserving and consistency-preserving protocols (AC protocols) in this section, and the policy-preserving and consistency-preserving protocols (PC protocols) in the appendix Section 13.

Leave Protocol. We first consider the Leave protocol presented in Algorithm 1, and then intuitively explain how it preserves the properties of the quorum system.

Algorithm 1 AC Leave and Remove

```
1  Implements: Leave and Remove
2    request : Leave | Remove(q)
3    response : LeaveComplete | LeaveFail
4               RemoveComplete | RemoveFail
5  Variables:
6    Q  ▷ Individual minimal quorums of self
7    tomb : 2^𝒫 ← ∅
8    (in-sink : Boolean, F : 2^𝒫) ← Discovery(Q)
9  Uses:
10   tob : TotalOrderBroadcast
11   apl : (∪Q) ∪ F ↦ AuthPPoint2PointLink
12 upon request Leave
13   if in-sink then
14     if ∀q1, q2 ∈ Q, (q1 ∩ q2)\{self} is
         self-blocking then
15       tob request Check(self, Q)
16     else
17       response LeaveFail
18   else
19     response LeaveComplete
20     apl(p) request Left(self)
         for each p ∈ F
21 upon response tob, Check(p′, Q′)
22   if ∃q1, q2 ∈ Q′. (q1 ∩ q2) \ ({p′} ∪ tomb)
       is not p′-blocking then
23     if p′ = self then
24       response LeaveFail
25   else
26     tomb ← tomb ∪ {p′}
27     if p′ = self then
28       response LeaveComplete
29       apl(p) request Left(self)
           for each p ∈ F
30 upon response apl(p), Left(p)
31   Q ← {q \ {p} | q ∈ Q}
32 upon request Remove(q)
33   ▷ Handlers for Remove are similar to Leave
     except: The quorum q that should be
     removed is passed in the Check message
     (instead of Q) at line L. 15, and the handler
     Check at L. 21 takes q as a parameter
     (instead of Q′). The update Q ← Q \ {q} is
     added after L. 28.
```

Variables and sub-protocols. Each process keeps its own set of individual minimal quorums Q. It also keeps the set *tomb* that records the processes that might have left. We saw that Lemma 20 presented an optimization opportunity for the coordination needed to preserve consistency: when the quorum system has quorum sharing, only processes in the sink component need coordination. Therefore, each process stores whether it is in the sink component as the *in-sink* boolean, and its follower processes (*i.e.*, processes that have this process in their quorums) as the set F. (Processes can use a sink discovery protocol such as the one we present in the appendix Section 15. The sink information is just used for an optimization, and the protocol can execute without it.)

The protocol uses a total-order broadcast *tob*, and authenticated point-to-point links *apl* (to processes in the quorums Q and followers F). Total-order broadcast provides a broadcast interface on top of consensus [50, 44, 30, 42]. The consensus and total-order broadcast abstractions [42] require quorum intersection for safety, and quorum availability and inclusion for liveness. As we will show, the reconfiguration protocols preserve both of these properties for outlived quorum systems. The total-order broadcast ensures the following safety property: for every pair of messages m and m', and well-behaved processes p and p', if m is delivered before m' at p, then at p', the message m' is either not delivered or delivered after m. Further, it ensures the following liveness property: every outlived process will eventually deliver every message that a well-behaved process sends. We note that if a protocol naively uses *tob* to globally order and process reconfigurations, then since each process only knows its own quorums, it cannot independently check if the properties of the quorum system are preserved.

Protocol. When a process requests to leave (at L. 12), it first checks whether it is in the sink component (at L. 13). If it is not in the sink, then by Lemma 20, it can apply the optimizations that are shown with the blue color. The process can simply leave without synchronization (at L. 19); it only needs to inform its follower set so that they can preserve their quorum availability. It sends a *Left* message to its followers (at L. 20). Every well-behaved process that receives the message (at L. 30) removes the sender from its quorums (at L. 31). If the quorum system does not have quorum sharing or the sink information is not available, the protocol can be conservative (remove the blue lines) and always perform the coordination that we will consider next.

On the other hand, when the requesting process is in the sink component, its absence can put quorum intersection in danger. Therefore, it first locally checks a condition (at L. 14). The check is just an optimization not to attempt leave requests that are locally known to fail. We will consider this condition in the next subsection. If the check fails, the leave request fails (at L. 17). If the local check passes, the process broadcasts a *Check* request together with its quorums (at L. 15). If processes receive and check concurrent leave requests in different orders, they may concurrently approve leave requests for all processes in a quorum intersection. Therefore, a total-order broadcast *tob* is used to enforce a total order for processing of *Check* messages. When a process receives a *Check* request with a set of quorums Q, it locally checks a condition for Q (at L. 22). This check is similar to the check above but is repeated in the total order of deliveries by the *tob*. If the condition fails, the leave request fails (at L. 24). If it passes, the leaving process is added to the *tomb* set (at L. 26), and the leaving process informs its followers, and leaves (at lines 28 and 29).

Intuition. Let's now consider the checked condition and see how it preserves quorum intersection and inclusion.

Quorum Intersection. Let us first see an intuitive explanation of the condition, and why it preserves quorum intersection. We assume that the quorum system is outlived: there is a set of processes $\mathcal{O}$ such that the quorum system has quorum intersection at $\mathcal{O}$, quorum inclusion for $\mathcal{O}$, and quorum availability inside $\mathcal{O}$. As shown in Figure 5, consider well-behaved processes p_1 and p_2 with quorums q_1 and q_2 respectively, and let p^* be a process at the intersection of q_1 and q_2 in $\mathcal{O}$. The goal is to allow p^* to leave only if the intersection of q_1 and q_2 contains another process in $\mathcal{O}$. By the quorum inclusion property, p^* should have quorums q_1^* and q_2^* such that their well-behaved processes are included inside q_1 and q_2 respectively. Each process adds to its *tomb* set every process whose *Check* request passes. The total-order-broadcast *tob* delivers the *Check* requests in the same order across processes. Therefore, the result of the check and the updated *tomb* set is the same across processes after processing each request. Consider a *Check* request of a process p' which is ordered before that of p^*. If the check for p' is passed and it leaves, then the *tomb* set of p^* contains p'. Consider when the *Check* request of p^* is processed. The check ensures that p^* is approved to leave only if the intersection of q_1^* and q_2^* modulo the *tomb* set and p^* is p^*-blocking. By Lemma 8, since the quorum system is available inside $\mathcal{O}$, this means that the intersection of q_1^* and q_2^* after both p' and p^* leave still intersects $\mathcal{O}$. A process p in $\mathcal{O}$ remains in the intersection of q_1^* and q_2^*. Therefore, by quorum inclusion, p remains in the intersection of q_1 and q_2. Thus, outlived quorum intersection is preserved for q_1 and q_2.

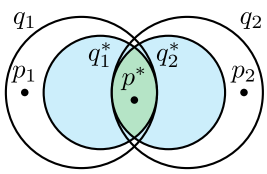


Figure 5 The Leave and Remove Protocols, Preserving Quorum Intersection.

Once the *tob* delivers the *Check* message of the leaving process p^* to p^* itself, it can locally decide whether it is safe to leave. We note that the local check ensures a global property: quorum intersection for the whole quorum system. We also note that both quorum inclusion and quorum availability are needed to preserve quorum intersection. Further, we note that outlived quorum intersection is not affected if a Byzantine process leaves: the outlived processes where quorums intersect are by definition a subset of well-behaved processes.

Quorum inclusion. Now let us elaborate on the quorum inclusion property that we just used. When a process p' leaves, it sends *Left* messages to its followers (at either L. 20 or L. 29). The followers later remove p' from their quorums (at L. 30-L. 31). These updates are not atomic and happen over time. Therefore, there might be a window when a process p' is removed from the quorum q_1 (that we saw above), but not yet removed from q_1^*. Therefore, quorum inclusion only eventually holds. However, we observe that in the meanwhile, a weaker notion of quorum inclusion, that we call *active quorum inclusion*, is preserved. It considers inclusion only for the active set of processes $\mathcal{A} = \mathcal{P} \setminus \mathcal{L}$, *i.e.*, it excludes the subset $\mathcal{L}$ of processes that have already left. It requires the quorum q_1^* to be a subset of q_1 modulo $\mathcal{L}$. More precisely, it requires $q_1^* \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1$. This weaker notion is enough to preserve quorum intersection. In the above discussion for quorum intersection, the process p that remains in the intersection is not in the *tomb* set; therefore, it is an active process. Since it is in q_1^* and q_2^*, by active quorum inclusion, it will be in q_1 and q_2 as well.

Remove Protocol. Let us now consider the Remove protocol. Removing a quorum can endanger all the three properties of the quorum system: inclusion, availability, and even intersection. Consider a process p that removes a quorum q. (1) Let p be an outlived process, and let p' be a well-behaved process with quorum q' that includes p and q, but no other quorum of p. The removal of q, violates outlived quorum inclusion for q'. (2) If q is the only quorum of p in the outlived set, the removal of q violates outlived availability for p. (3) As we saw above, p can lose outlived availability, and fall out of the outlived set. Consider a pair of quorums whose intersection includes only p from the outlived set. The removal of q violates outlived quorum intersection for these pair of quorums. Therefore, similar to a leaving process, a process that removes a quorum should coordinate, check the safety of its reconfiguration, and update others' quorums. As shown in Algorithm 1, the Remove protocol is, thus, similar to the Leave protocol. The difference is that when a request to remove a quorum q is successful, q is removed from the quorums of the requesting process (after L. 28).

Correctness. The Leave and Remove protocols maintain the properties of the quorum system. We prove that they preserve quorum intersection, and eventually provide quorum availability and quorum inclusion.

Let $\mathcal{L}$ denote the set of processes that receive a *LeaveComplete* or *RemoveComplete* response. As we saw before, processes in $\mathcal{L}$ may fall out of the outlived set. Starting from a quorum system that is outlived for $\mathcal{O}$, the protocols only eventually result in an outlived quorum system for $\mathcal{O} \setminus \mathcal{L}$. However, they preserve strong enough notions of quorum inclusion and availability, called active quorum inclusion and active quorum availability, which support quorum intersection to be constantly preserved. Consider a quorum q of p, and a process p' of q that falls in $\mathcal{L}$. Intuitively, active quorum inclusion for q does not require the inclusion of a quorum of p' in q, and active availability for p does not require p' to be well-behaved. We first capture these weaker notions and prove that they are preserved, and further, prove that quorum inclusion and availability eventually hold. We then use the above two preserved properties to prove that the protocols preserve quorum intersection at $\mathcal{O} \setminus \mathcal{L}$. The correctness theorems and proofs are available in section Section 12 and Section 16.

7 Add

We saw the trade-off for the add operation in Theorem 22. Since we never sacrifice consistency, we present an Add protocol that preserves consistency and availability. For brevity, we present an intuition and summary of the protocol in this section.

Example. Let us first see how adding a quorum for a process can violate the quorum inclusion and quorum intersection properties. Consider our running example from Figure 1. As we saw before, the outlive set is $\mathcal{O} = \{2, 3, 5\}$. If 3 adds a new quorum $\{3, 5\}$ to its set of quorums, it violates quorum inclusion for $\mathcal{O}$. The new quorum includes process 5 that is outlive. However, process 5 has only one quorum $\{2, 5\}$ that is not a subset of $\{3, 5\}$. Further, quorum intersection is violated since the quorum $\{1, 2, 4\}$ of 1 does not have a well-behaved intersection with $\{3, 5\}$.

Consider a quorum system that is outlive for a set of well-behaved processes $\mathcal{O}$, and a well-behaved process p that wants to add a new quorum q_n. (If the requesting process p is Byzantine, it can trivially add any quorum. Further, we consider new quorums q_n that have at least one well-behaved process. Otherwise, no operation by the quorum is credible. For example, p itself can be a member of q_n.)

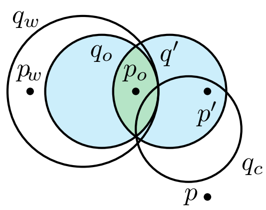


Figure 6 The Add protocol, Preserving Quorum Intersection.

Intuition. Now, we explain the intuition of how the quorum inclusion and quorum intersection properties are preserved, and then an overview of the protocol.

Quorum inclusion. In order to preserve quorum inclusion, process p first asks each process in q_n whether it already has a quorum that is included in q_n. It gathers the processes that respond negatively in a set q_c. (In this overview, we consider the main case where there is at least one well-behaved process in q_c. The other cases are straightforward, and discussed in the proof of Lemma 38.) To ensure quorum inclusion, the protocol adds q_c as a quorum to every process p' in q_c. Since these additions do not happen atomically, quorum inclusion is only eventually restored. In order to preserve a weak notion of quorum inclusion called tentative quorum inclusion, each process stores a *tentative* set of quorums, in addition to its set of quorums. The protocol performs the following actions in order. It first adds q_c to the *tentative* quorums of every process in q_c, then adds q_n to the quorums of p, then adds q_c to the quorums of every process in q_c, and finally garbage-collects q_c from the *tentative* sets. Thus, the protocol preserves tentative quorum inclusion: for every quorum q, and outlive process p in q, there is either a quorum *or a tentative quorum* q' of p such that well-behaved processes of q' are included in q. We will see that when a process performs safety checks, it considers its tentative quorums in addition to its quorums.

Quorum Intersection. Existing quorums in the system have outlive quorum intersection, *i.e.*, quorum intersection at $\mathcal{O}$. We saw that when a process p wants to add a new quorum q_n, a quorum $q_c \subseteq q_n$ may be added as well. We need to ensure that q_c is added only if outlive quorum intersection is preserved. We first present the design intuition, and then an overview of the steps of the protocol.

The goal is to approve adding q_c as a quorum only if it has an outlive intersection with every other quorum q_w in the system. Lemma 8 presents an interesting opportunity to check this condition locally. It states that if an outlive process finds a set self-blocking, then that set has an outlive process. Thus, if we can pass the quorum q_c to an outlive process p_o, and have it check that $q_c \cap q_w$ is self-blocking, then we have that the intersection of q_c and

q_w has an outlive process. However, no outlive process is aware of all quorums q_w in the system. Tentative quorum inclusion can help here. If the outlive process p_o is inside p_w, then by outlive quorum inclusion, p_o has a quorum or tentative quorum q_o (whose well-behaved processes are) included in p_w. The involved quorums are illustrated in Figure 7. If the outlive process p_o check for all its own quorums or tentative quorums q (including q_o), that $q_c \cap q$ is self-blocking then, by Lemma 8, the intersection of q_c and q_o has an outlive process. Since q_o is included in q_w, we get the desire result that the intersection of q_c and q_w has an outlive process. However, how do we reach from the requesting process p to an outlive process p_o in every quorum q_w in the system? Process p that is requesting to add the quorum q_c doesn't know whether it is outlive itself. There is at least a well-behaved process p' in the quorum q_c. By outlive quorum intersection, every quorum q' of p' intersects with every other quorum q_w at an outlive process p_o. Therefore, the protocol takes two hops to reach to the outlive process p_o: process p asks the processes p' of q_c, and then p' asks the processes p_o in its quorums q'.

Based on the intuition above, we now consider an overview of the protocol. Before adding q_c as a quorum for each process in q_c, the protocol goes through two hops. In the first hop, process p asks each process p' in q_c to perform a check. In the second hop, process p' asks its quorums q' to perform a check. A process p_o in q' checks for every quorum q_o in its set of quorums and tentative quorums that $q_o \cap q_c$ is self-blocking. If the check passes, p_o sends an Ack to p'; otherwise, it sends a Nack. Process p' waits for an Ack from at least one of its quorums q' before sending a commit message back to p. Once p receives a commit message from each process in q_c, it safely adds q_n to its own set of quorums, and requests each process in q_c to add q_c as a quorum.

In the limited space, we presented an overview of the add protocol. The details of the protocol and its correctness proofs are available in the appendix Section 14 and Section 17.

8 Sink Discovery

Following the graph characterization that we saw in Section 4, we now present a decentralized protocol that can find whether each process is in the sink component of the quorum graph. We first describe the protocol and then its properties.

Protocol. Consider a quorum system with quorum intersection, availability and sharing. The sink discovery protocol in Algorithm 8 finds whether each well-behaved process is in the sink. It also finds the set of its followers. A process p is a follower of process p' iff p has a quorum that includes p'. The protocol has two phases. In the first, it finds the well-behaved minimal quorums, *i.e.*, every minimal quorum that is a subset of well-behaved processes. Since well-behaved minimal quorums are inside the sink, the second phase extends the discovery to other processes in the same strongly connected component.

Variables and sub-protocols. In the quorum system $\mathcal{Q}$, each process **self** stores its own set of individual minimal quorums $Q = \mathcal{Q}(\textbf{self})$, a map *qmap* from other processes to their quorums which is populated as processes communicate, the *in-sink* boolean that stores whether the process is in the sink, and the set of follower processes F. The protocol uses authenticated point-to-point links *apl*. They provide the following safety and liveness properties. If the sender and receiver are both well-behaved, then the message will be eventually delivered. Every message that a well-behaved process delivers from a well-behaved process is sent by the later, *i.e.*, the identity of a well-behaved process cannot be forged.

Protocol. When processes receive a *Discover* request (at L. 8), they exchange their quorums with each other. In this first phase, each process sends an *Exchange* message with its quorums Q to all processes in its quorums. When a process receives an *Exchange* message (at L. 10), it adds the sender to the follower set F, and stores the received quorums in its *qmap*. As *qmap* is populated, when a process finds that one of its quorums q is a quorum of every other process in q as well (at L. 13), by Lemma 12, it finds that its quorum q is a minimal quorum, and by Theorem 19, it finds itself in the sink. Thus, it sets its *in-sink* variable to true in the first phase (at L. 14). The process then sends an *Extend* message with the quorum q to all processes of its own quorums Q (at L. 15). The *Extend* messages are processed in the second phase. The processes of every well-behaved minimal quorum are found in this phase. In Figure 3, since the quorum $\{1, 2\}$ is a quorum for both of 1 and 2, they find themselves in the sink. However, process 3 might receive misleading quorums from process 5, and hence, may not find itself in the sink in this phase.

The processes P_1 of every well-behaved minimal quorum find themselves to be in the sink in the first phase. Let P_2 be the well-behaved processes of the remaining minimal quorums. A pair of minimal quorums have at least a well-behaved process in their intersection. In Figure 3, the two minimal quorums $P_1 = \{1, 2\}$ and $P_2 = \{1, 3, 5\}$ intersect at 1. Therefore, by Lemma 13, every process in P_2 is a neighbor of a process in P_1. Thus, in the second phase, the processes P_1 can send *Extend* messages to processes in P_2, and inform them that they are in the sink. In Figure 3, process 1 can inform process 3. The protocol lets a process accept an *Extend* message containing a quorum q only when the same message comes from the intersection of q and one of its own quorums q' (at L. 16). Let us see why a process in P_2 cannot accept an *Extend* message from a single process. A minimal quorum q that is found in phase 1 can have a Byzantine process p_1. Process p_1 can send an *Extend*(q) message (even with signatures from members of q) to a process p_2 even if p_2 is not a neighbor of p_1, and make p_2 believe that it is in the sink. In Figure 3, the Byzantine process 5 can collect the quorum $\{1, 3, 5\}$ from 1 and 3, and then send an *Extend* message to 4 to make 4 believe that it is inside the sink. Therefore, a process p_2 in P_2 accepts an *Extend*(q) message only when it is received from the intersection of q and one of its own quorums. Since there is a well-behaved process in the intersection of the two quorums, process p_2 can then trust the *Extend* message. When the check passes, p_2 finds itself to be in the sink, and sets the *in-sink* variable to true in the second phase (at L. 17). In Figure 3, when process 3 receives an *Extend* message with quorum $\{1, 2\}$ from 1, since $\{1\}$ is the intersection of the quorum $\{1, 3, 5\}$ of 3, and the received quorum $\{1, 2\}$, process 3 accepts the message.

Let *ProtoSink* denote the set of well-behaved processes where the protocol sets the *in-sink* variable to true. The discovery protocol is complete: all the well-behaved processes of minimal quorums will eventually know that they are in the sink (*i.e.*, set their *in-sink* to true).

▶ **Lemma 23** (Completeness). *For all $q \in MQ(\mathcal{Q})$, eventually $q \cap \mathcal{W} \subseteq$ ProtoSink.*

This result brings an optimization opportunity: the leave and remove protocols can coordinate only when a process inside *ProtoSink* is updated. Although, completeness is sufficient for safety of the optimizations, in the appendix Section 18, we prove both the completeness and accuracy of the two phases in turn. The accuracy property states that *ProtoSink* is a subset of the sink component.

9 Related Works

Quorum Systems with Heterogeneous Trust. We described a few instances of heterogeneous quorum systems in Section 3. The blockchain technology raised the interest in quorum systems that allow non-uniform trust preferences for participants, and support open

Algorithm 2 Sink Discovery Protocol

Variables:
  Q ▷ The individual minimal quorums of **self**
  $qmap : \mathcal{P} \mapsto \mathsf{Set}[2^{\mathcal{P}}]$
  $\textit{in-sink} : \mathsf{Boolean} \leftarrow \mathsf{false}$
  $F : 2^{\mathcal{P}}$
Uses:
  $apl : \mathcal{P} \mapsto \mathsf{AuthPerfectPointToPointLink}$
upon request $\textit{Discover}$
  $apl(p)$ **request** $\textit{Exchange}(Q)$ for each $p \in \cup Q$
upon response $apl(p)$, $\textit{Exchange}(Q')$
  $F \leftarrow F \cup \{p\}$
  $qmap(p) \leftarrow Q'$
upon $\exists q \in Q.\ \forall p \in q.\ q \in qmap(p)$
  $\textit{in-sink} \leftarrow \mathsf{true}$
  $apl(p)$ **request** $\textit{Extend}(q)$ for each $p \in \cup Q$
upon response $\overline{apl(p)}, \textit{Extend}(q)$ s.t.$\exists q' \in Q.\ \{\overline{p}\} = q \cap q'$
  $\textit{in-sink} \leftarrow \mathsf{true}$

admission and release of participants. Ripple [58] and Cobalt [46] pioneered decentralized admission. They let each node specify a list, called the unique node list (UNL), of processes that it trusts. However, they assume that 60-90% of every pair of lists overlap. It has been shown that violation of this assumption can compromise the security of the network [6, 64], further highlighting the importance of formal models and proofs [34, 11].

Stellar [50] provides a consensus protocol for federated Byzantine quorum systems (FBQS) [29, 30] where nodes are allowed to specify sets of processes, called slices, that they trust. The Stellar system [43] uses hierarchies and thresholds to specify quorum slices and provides open membership. Since each process calculates its own quorums from slices separately, the resulting quorums do not necessarily intersect, and after independent reconfigurations "the remaining sets may not overlap, which could cause network splits" [25]. Therefore, to prevent forks, a global intersection check is continually executed over the network. Follow-up research analyzed the decentralization extent of Stellar [12, 37], and discussed [28] reconfiguration for the uniform quorums of the top tier nodes. This paper presents reconfiguration protocols that preserve the safety of heterogeneous quorum systems.

Personal Byzantine quorum systems (PBQS) [44] capture the quorum systems that FBQSs derive from slices, require quorum intersection only inside subsets of processes called clusters, and propose a consensus protocol. It defines the notion of quorum sharing. As we saw in Section 3, this paper presents quorum inclusion that is weaker than quorum sharing; therefore, a cluster is outlived but not vice versa. This paper showed that quorum inclusion is weak enough to be preserved during reconfiguration, and strong enough to support preserving consistency and availability.

Flexible BFT [48] allows different failure thresholds between learners. Heterogeneous Paxos [59, 60] further generalizes the separation between learners and acceptors with different trust assumptions. Further, it specifies quorums as sets rather than number of processes. These two projects introduce consensus protocols. However, they require the knowledge of all processes in the system. In contrast, this paper presents HQSs that requires only local knowledge, captures their properties, and presents reconfiguration protocols for them.

Asymmetric trust [24] lets each process specify the sets of processes that its doesn't trust, and considers broadcast, secret-sharing, and multi-party computation problems. Similarly, in asymmetric Byzantine quorum systems (ABQS) [18, 19, 4] each process defines its subjective dissemination quorum system (DQS): in addition to its sets of quorums, each process specifies sets of processes that it believes may mount Byzantine attacks. This work presents shared memory and broadcast protocols, and further, rules to compose two ABQSs. The followup model [17] lets each process specify a subjective DQS for processes that it knows, transitively relying on the assumptions of other processes. On the other hand, this paper presents decentralized reconfiguration protocols to add and remove processes and quorums. Further, it lets each process specify only its own set of quorums, and captures the properties of the resulting quorum systems.

Multi-threshold [35] and MT-BFT [52] broadcast protocols elaborate Bracha [13] to have different fault thresholds for different properties and for different synchrony assumptions but have uniform quorums. K-CRB [10] supports non-uniform quorums and delivers up to k different messages.

Quorum subsumption [42] adopted and cited HQS from the arXiv version of this paper, and presented a generalization of quorum sharing called quorum subsumption. This paper focuses on maintaining the properties of HQS when processes perform reconfigurations. We found quorum inclusion as a flavor of quorum sharing that is weak enough to be maintained during reconfiguration, and strong enough to support the consistency and availability properties. In fact, quorum inclusion is weaker than quorum subsumption: for a quorum q, it requires only the processes of q that are in P to have a quorum q', and only the well-behaved part of q' to be a subset of q.

Open Membership. We consider three categories.

Group Membership. As processes leave and join, group membership protocols [23] keep the same global view of members across the system (although the set of all processes may be fixed). Pioneering work, Rambo [45] provides atomic memory, and supports join and leave reconfigurations. It uses Paxos [39] to totally order reconfiguration requests, and tolerates crash faults. Rambo II [32] improves latency by garbage collecting in parallel. Recently, multi-shard atomic commit protocols [14] reduce the number of replicas for each shard and reconfigure the system upon failures. Since accurate membership is as strong as consensus [22, 23], classical [57] and recent Byzantine group membership protocols such as Cogsworth [54] and later works [31] use consensus to reach an agreement on membership and adjust quorums accordingly. Recent works present more abstractions on top of group membership: DBRB [33] provides reliable broadcast, DBQS [5] preserves consistent read and write quorums, and further adjusts the cardinality of quorums according to the frequency of failures, Dyno [26] provides replication, and SmartMerge [36] provides replication, and uses a commutative merge function on reconfiguration requests to avoid consensus. Existing protocols consider only cardinality-based or symmetric quorum systems where quorums are uniform across processes. On the other hand, this paper presents reconfiguration protocols for heterogeneous quorum systems.

Hybrid Open Membership. Solida [1], Hybrid Consensus [56], Tendermint [15, 7], Casper [16], OmniLedger [38], and RapidChain [66] blockchains combine permissionless and permissioned replication [47] to provide both consistency and open membership [9]. They use permissionless consensus to dynamically choose validators for permissioned consensus.

Unknown participants and network topology. BCUP and BFT-CUP [21, 3, 2] consider consensus in environments with unknown participants. They assume properties about the topology and connectivity of the network, and consider only uniform quorums. Later, [55] presents necessary and sufficient conditions for network connectivity and synchrony for consensus in the presence of crash failures and flaky channels. In contrast, this paper considers reconfiguration for Heterogeneous Byzantine quorums, and presents optimizations based on the quorum topology.

10 Conclusion

This paper presents a model of heterogeneous quorum systems, their properties, and their graph characterization. In order to make them open, it addresses their reconfiguration. It proves trade-offs for the properties that reconfigurations can preserve, and presents reconfiguration protocols with provable guarantees, We hope that this work further motivates the incorporation of open membership and heterogeneous trust into quorum systems, and helps blockchains avoid high energy consumption, and centralization at nodes with high computational power or stake.

References

1. Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. *arXiv preprint arXiv:1612.02916*, 2016.
2. Eduardo Adilio Pelinson Alchieri, Alysson Bessani, Fabiola Greve, and Joni da Silva Fraga. Knowledge connectivity requirements for solving byzantine consensus with unknown participants. *IEEE Transactions on Dependable and Secure Computing*, 15(2):246–259, 2016.
3. Eduardo AP Alchieri, Alysson Neves Bessani, Joni da Silva Fraga, and Fabíola Greve. Byzantine consensus with unknown participants. In *International Conference On Principles Of Distributed Systems*, pages 22–40. Springer, 2008.
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. Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, Michael K Reiter, and Rebecca N Wright. Dynamic byzantine quorum systems. In *Proceeding International Conference on Dependable Systems and Networks. DSN 2000*, pages 283–292. IEEE, 2000.
6. Ignacio Amores-Sesar, Christian Cachin, and Jovana Mićić. Security analysis of ripple consensus. *arXiv preprint arXiv:2011.14816*, 2020.
7. Yackolley Amoussou-Guenou, Antonella Del Pozzo, Maria Potop-Butucaru, and Sara Tucci-Piergiovanni. Correctness of tendermint-core blockchains. In *22nd International Conference on Principles of Distributed Systems (OPODIS 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.
8. Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the thirteenth EuroSys conference*, pages 1–15, 2018.
9. Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. Sok: Consensus in the age of blockchains. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 183–198, 2019.
10. 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.
11. Ranadeep Biswas, Michael Emmi, and Constantin Enea. On the complexity of checking consistency for replicated data types. In *Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II 31*, pages 324–343. Springer, 2019.

12 Andrea Bracciali, Davide Grossi, and Ronald de Haan. Decentralization in open quorum systems: Limitative results for ripple and stellar. In *2nd International Conference on Blockchain Economics, Security and Protocols (Tokenomics 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

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

14 Manuel Bravo and Alexey Gotsman. Reconfigurable atomic transaction commit. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 399–408, 2019.

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

16 Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *arXiv preprint arXiv:1710.09437*, 2017.

17 Christian Cachin, Giuliano Losa, and Luca Zanolini. Quorum systems in permissionless network. *OPODIS 2022: 17:1-17:22*, 2022.

18 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.

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

20 Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

21 David Cavin, Yoav Sasson, and André Schiper. Consensus with unknown participants or fundamental self-organization. In *International Conference on Ad-Hoc Networks and Wireless*, pages 135–148. Springer, 2004.

22 Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 322–330, 1996.

23 Gregory V Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

24 Ivan Damgård, Yvo Desmedt, Matthias Fitzi, and Jesper Buus Nielsen. Secure protocols with asymmetric trust. In *Advances in Cryptology–ASIACRYPT 2007: 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007. Proceedings 13*, pages 357–375. Springer, 2007.

25 Stellar Developer. Stellar configuring. https://developers.stellar.org/docs/run-core-node/configuring, 2022.

26 Sisi Duan and Haibin Zhang. Foundations of dynamic bft. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1546–1546. IEEE Computer Society, 2022.

27 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

28 Martin Florian, Sebastian Henningsen, Charmaine Ndolo, and Björn Scheuermann. The sum of its parts: Analysis of federated byzantine agreement systems. *Distributed Computing*, pages 1–19, 2022.

29 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.

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

31 Rati Gelashvili, Lefteris Kokoris-Kogias, Alexander Spiegelman, and Zhuolun Xiang. Brief announcement: Be prepared when network goes bad: An asynchronous view-change protocol. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*, pages 187–190, 2021.

32 Seth Gilbert. *RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks*. PhD thesis, Massachusetts Institute of Technology, 2003.

33 Rachid Guerraoui, Jovan Komatovic, Petr Kuznetsov, Yvonne-Anne Pignolet, Dragos-Adrian Seredinschi, and Andrei Tonkikh. Dynamic byzantine reliable broadcast [technical report]. *arXiv preprint arXiv:2001.06271*, 2020.

34 Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.

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

36 Leander Jehl, Roman Vitenberg, and Hein Meling. Smartmerge: A new approach to reconfiguration for atomic storage. In *International Symposium on Distributed Computing*, pages 154–169. Springer, 2015.

37 Minjeong Kim, Yujin Kwon, and Yongdae Kim. Is stellar as secure as you think? In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 377–385. IEEE, 2019.

38 Eleftherios Kokoris-Kogias, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ewa Syta, and Bryan Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 583–598. IEEE, 2018.

39 Leslie Lamport. Paxos made simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pages 51–58, 2001.

40 Andrew Lewis-Pye and Tim Roughgarden. Byzantine generals in the permissionless setting. *arXiv preprint arXiv:2101.07095*, 2021.

41 Andrew Lewis-Pye and Tim Roughgarden. Permissionless consensus. *arXiv preprint arXiv:2304.14701*, 2023.

42 Xiao Li, Eric Chan, and Mohsen Lesani. Quorum subsumption for heterogeneous quorum systems. In *37th International Symposium on Distributed Computing (DISC 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2023.

43 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.

44 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.

45 Nancy Lynch and Alex A Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Distributed Computing: 16th International Conference, DISC 2002 Toulouse, France, October 28–30, 2002 Proceedings 16*, pages 173–190. Springer, 2002.

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

47 Ivan Malakhov, Andrea Marin, Sabina Rossi, and Daria Smuseva. On the use of proof-of-work in permissioned blockchains: Security and fairness. *IEEE Access*, 10:1305–1316, 2021.

48 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.

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

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

51 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.

52 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.

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

54 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. *arXiv preprint arXiv:1909.05204*, 2019.

55 Alejandro Naser-Pastoriza, Gregory Chockler, and Alexey Gotsman. Fault-tolerant computing with unreliable channels. In *27th International Conference on Principles of Distributed Systems (OPODIS 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2024.

56 Rafael Pass and Elaine Shi. Hybrid consensus: Efficient consensus in the permissionless model. *Cryptology ePrint Archive*, 2016.

57 Michael K Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, 1996.

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

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

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

61 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.

62 Chrysoula Stathakopoulou, Tudor David, and Marko Vukolic. Mir-bft: High-throughput bft for blockchains. *arXiv preprint arXiv:1906.05552*, page 92, 2019.

63 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.

64 Levin N Winter, Florena Buse, Daan De Graaf, Klaus Von Gleissenthall, and Burcu Kulahcioglu Ozkan. Randomized testing of byzantine fault tolerant algorithms. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):757–788, 2023.

65 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.

66 Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. Rapidchain: A fast blockchain protocol via full sharding. *IACR Cryptol. ePrint Arch.*, 2018:460, 2018.

Appendix

Contents

11 Join

Algorithm 3 Join

1 **Implements:** Join
2 **request** : $Join(ps)$
3 **response** : $JoinComplete$
4 **Variables:**
5 Q : Set$[2^{\mathcal{P}}]$ ▷ Individual minimal quorums of **self**
6 S : Set$[2^{\mathcal{P}}]$
7 F : Set$[\mathcal{P}]$ ▷ Followers
8 $qmap : \mathcal{P} \mapsto$ Set$[2^{\mathcal{P}}]$
9 **Uses:**
10 $apl : \mathcal{P} \mapsto$ AuthPerfectPointToPointLink
11 **upon request** $Join(ps)$
12 $S \leftarrow \{\{q\}\}, qmap \leftarrow \emptyset$
13 **upon** $\exists q \in S, \exists p \in q$, s.t. $qmap(p) = \emptyset$
14 $apl(p)$ **request** $Prob$
15 **upon response** $apl(p')$, $Prob$
16 $F \leftarrow F \cup \{p'\}$
17 $apl(p')$ **request** $Quorums(Q)$
18 **upon response** $apl(p')$ $Quorums(Q')$
19 $qmap \leftarrow qmap[p' \mapsto Q']$
20 **for each** $q \in S$ s.t. $p' \in q$
21 **for each** $q' \in Q'$
22 $S \leftarrow S \setminus \{q\} \cup \{q \cup q'\}$
23 **upon** $\forall q \in S.\ \forall p \in q.\ \exists q' \in qmap(p).\ q' \subseteq q$
24 $Q \leftarrow S$
25 **response** $JoinComplete$

We now consider the Join protocol which is presented in Algorithm 3. When a process p wants to join the system, it issues a *Join* request with an initial set of processes *ps*, which is a set of processes that it trusts (at L. 11). In order to maintain the quorum inclusion property, the requesting process starts with *ps* as a tentative quorum and probes these processes for their quorums (at L. 13-L. 14). When a process receives a probe request, it sends back it's quorums, and adds the sender to its follower set (at L. 15-L. 17). When a quorum from a process p is received, it is added to each tentative quorum that contains p (at L. 18-L. 22). The tentative quorums grow and probing continues for the new members. It stops when the tentative quorums are quorum including (at L. 23).

Correctness. We now show that the Join protocol preserves all the three properties of the quorum system.

▶ **Lemma 24.** *For every quorum system and well-behaved set of processes $\mathcal{O}$, the* Join *protocol preserves quorum intersection at $\mathcal{O}$, quorum availability for $\mathcal{O}$, and quorum inclusion for $\mathcal{O}$. Therefore, it preserves every outlived set for the quorum system. Further, newly joined processes have quorum inclusion.*

Proof. Quorum intersection at $\mathcal{O}$ is preserved since the existing quorums have intersection at $\mathcal{O}$, and the new quorums are supersets of existing quorums. Quorum availability and quorum inclusion for $\mathcal{O}$ are preserved since the quorums of existing processes do not change. Therefore, by the three properties above, it preserves every outlived set $\mathcal{O}$. Further, a newly joined process has quorum inclusion, since the new quorums pass the condition at L. 23 which is sufficient for quorum inclusion. ◀

We add that for adding a quorum, if adding a superset of the given quorum is considered policy-preserving, then the add protocol can be similar to the join protocol above.

12 AC Leave and Remove

12.1 Correctness

We now state that Leave and Remove protocols maintain the properties of the quorum system. We prove that they preserve quorum intersection, and eventually provide quorum availability and quorum inclusion.

Starting from an outlived quorum system, the protocols only eventually result in an outlived quorum system. However, they preserve strong enough notions of quorum inclusion and availability, called active quorum inclusion and active quorum availability, which support quorum intersection to be constantly preserved. We first capture these weaker notions and prove that they are preserved, and further, prove that quorum inclusion and availability eventually hold. We then use the above two preserved properties to prove that quorum intersection is preserved. All in all, we show that the protocols maintain quorum intersection as a safety property, and quorum availability and inclusion as liveness properties.

Let $\mathcal{L}$ denote the set of processes that have received a *LeaveComplete* or *RemoveComplete* response. As we saw before, processes in $\mathcal{L}$ may fall out of the outlived set.

Quorum inclusion. Active quorum inclusion captures inclusion modulo the set $\mathcal{L}$.

▶ **Definition 25** (Active quorum inclusion). *A quorum system $\mathcal{Q}$ has* active *quorum inclusion for P iff for all well-behaved processes p and quorums q of p, if a process p' in q is inside P, then there is a quorum q' of p' such that well-behaved and active processes of q' are a subset of q i.e., $\forall p \in \mathcal{W}.\ \forall q \in \mathcal{Q}(p).\ \forall p' \in q \cap P.\ \exists q' \in \mathcal{Q}(p').\ q' \cap \mathcal{W} \setminus \mathcal{L} \subseteq q$.*

It is obvious that quorum inclusion implies active quorum inclusion. We now state that active quorum inclusion is preserved, and quorum inclusion is eventually reconstructed.

▶ **Lemma 26** (Preservation of Quorum inclusion). *The AC* Leave *and* Remove *protocols preserve active quorum inclusion. Further, starting from a quorum system that has quorum inclusion for processes $\mathcal{O}$, the protocols eventually result in a quorum system with quorum inclusion for $\mathcal{O} \setminus \mathcal{L}$.*

Quorum Availability. Let's define the notions of active availability and active blocking sets.

▶ **Definition 27** (Active Availability). *A quorum system has active availability inside a set of processes P iff every process p in $P \setminus \mathcal{L}$ has at least a quorum q such that $q \setminus \mathcal{L}$ is in P.*

It is obvious that availability implies active availability.

▶ **Definition 28** (Active Blocking Set). *A set of processes P is an active blocking set for a process p iff for every quorum q of p, the set $q \setminus \mathcal{L}$ intersects P.*

▶ **Lemma 29.** *For every quorum system that has active availability inside P, every active blocking set of every process in P intersects $P \setminus \mathcal{L}$.*

The proof is similar to the proof of Lemma 8. We use this lemma to show that active availability is preserved, and availability is eventually reconstructed.

▶ **Lemma 30** (Preservation of Availability). *The AC* Leave *and* Remove *protocols preserve active availability. Further, starting from a quorum system that has availability inside processes $\mathcal{O}$, the protocols eventually result in a quorum system with availability inside $\mathcal{O} \setminus \mathcal{L}$.*

Quorum Intersection. We saw that the protocols preserve active availability and active quorum inclusion. We use these two properties to show that they preserve quorum intersection.

▶ **Lemma 31** (Preservation of Quorum Intersection). *If a quorum system has quorum intersection at processes $\mathcal{O}$, active availability inside $\mathcal{O}$, and active quorum inclusion for $\mathcal{O}$, then the AC* Leave *and* Remove *protocols preserve quorum intersection at $\mathcal{O} \setminus \mathcal{L}$.*

Outlive. The three lemmas that we saw show that an outlived quorum system is eventually reconstructed.

▶ **Lemma 32** (Preservation of Outlived set). *Starting from a quorum system that is outlived for processes $\mathcal{O}$, the AC* Leave *and* Remove *protocols eventually result in a quorum system that is outlived for $\mathcal{O} \setminus \mathcal{L}$.*

Immediate from Lemma 31, Lemma 26, and Lemma 30.

The proofs are available in the appendix Section 16.

13 PC Leave and Remove

In this section, we present the Leave and Remove protocols that preserve both consistency and policies. We first consider the protocols before the correctness theorems.

Algorithm 4 PC Leave and Remove a quorum

```
1  Implements: Leave and Remove
2    request : Leave | Remove(q)
3    response : LeaveComplete | RemoveComplete
4  Variables:
5    Q                    ▷ The individual minimal quorums of self
6    F : Set[𝒫]
7  Uses:
8    apl : (∪Q) ∪ F ↦ AuthPerfectPointToPointLink
9  upon request Leave
10     apl(p) request Left(self)  for each p ∈ F
11     response LeaveComplete
12 upon response apl(p), Left(p)
13     Q ← Q \ {q ∈ Q | p ∈ q}
14 upon request Remove(q)
15     Q ← Q \ {q}
16     response RemoveComplete
```

The Leave and Remove protocols that preserve the policies are shown in Algorithm 4.

Variables and sub-protocols. Each process keeps its own set of individual minimal quorums Q. It also stores its follower processes (*i.e.*, processes that have this process in their quorums) as the set F.

The protocol uses authenticated point-to-point links *apl* (to each quorum member and follower).

Protocol. A process that requests a *Leave* informs its follower set by sending a *Left* message (at L. 10). Every well-behaved process that receives a *Left* message (at L. 12) removes any quorum that contains the sender (at L. 13) so that quorum intersection is not lost in case the intersection is the leaving process. A process that requests a *Remove*(q) simply removes q locally from its quorums Q.

Correctness. The protocols are both consistency- and policy-preserving.

▶ **Lemma 33.** *The PC* Leave *and* Remove *protocols are consistency-preserving.*

This is immediate from the fact that the protocols only remove quorums, and further for the leave protocol, the remaining quorums do not include the leaving process. Therefore, quorum intersection persists.

▶ **Lemma 34.** *The PC* Leave *and* Remove *protocols are policy-preserving.*

This is straightforward as the protocols remove but do not shrink quorums.

14 Add

We saw the trade-off for the add operation in Theorem 22. Since we never sacrifice consistency, we present an Add protocol that preserves consistency and availability.

Example. Let us first see how adding a quorum for a process can violate the quorum inclusion and quorum intersection properties. Consider our running example from Figure 1. As we saw before, the outlived set is $\mathcal{O} = \{2, 3, 5\}$. If 3 adds a new quorum $\{3, 5\}$ to its set of quorums, it violates quorum inclusion for $\mathcal{O}$. The new quorum includes process 5 that is outlive. However, process 5 has only one quorum $\{2, 5\}$ that is not a subset of $\{3, 5\}$. Further, quorum intersection is violated since the quorum $\{1, 2, 4\}$ of 1 does not have a well-behaved intersection with $\{3, 5\}$.

Consider a quorum system that is outlived for a set of well-behaved processes $\mathcal{O}$, and a well-behaved process p that wants to add a new quorum q_n. (If the requesting process p is Byzantine, it can trivially add any quorum. Further, we consider new quorums q_n that have at least one well-behaved process. Otherwise, no operation by the quorum is credible. For example, p itself can be a member of q_n.)

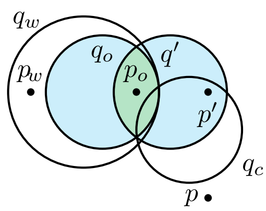


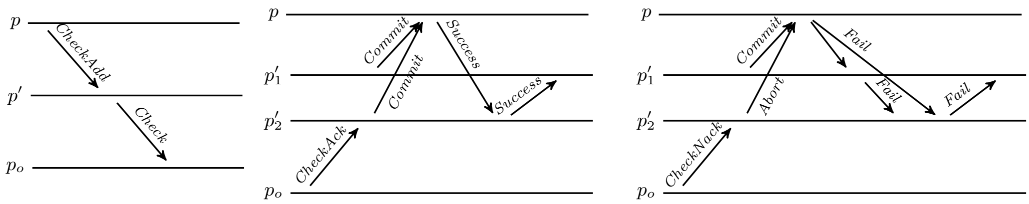
Figure 7 The Add protocol, Preserving Quorum Intersection.

Intuition. Now, we explain the intuition of how the quorum inclusion and quorum intersection properties are preserved, and then an overview of the protocol.

Quorum inclusion. In order to preserve quorum inclusion, process p first asks each process in q_n whether it already has a quorum that is included in q_n. It gathers the processes that respond negatively in a set q_c. (In this overview, we consider the main case where there is at least one well-behaved process in q_c. The other cases are straightforward, and discussed in the proof of Lemma 38.) To ensure quorum inclusion, the protocol adds q_c as a quorum to every process p' in q_c. Since these additions do not happen atomically, quorum inclusion is only eventually restored. In order to preserve a weak notion of quorum inclusion called tentative quorum inclusion, each process stores a *tentative* set of quorums, in addition to its set of quorums. The protocol performs the following actions in order. It first adds q_c to the *tentative* quorums of every process in q_c, then adds q_n to the quorums of p, then adds q_c to the quorums of every process in q_c, and finally garbage-collects q_c from the *tentative* sets. Thus, the protocol preserves tentative quorum inclusion: for every quorum q, and outlived process p in q, there is either a quorum *or a tentative quorum* q' of p such that well-behaved processes of q' are included in q. We will see that when a process performs safety checks, it considers its tentative quorums in addition to its quorums.

Quorum Intersection. Existing quorums in the system have outlived quorum intersection, *i.e.*, quorum intersection at $\mathcal{O}$. We saw that when a process p wants to add a new quorum q_n, a quorum $q_c \subseteq q_n$ may be added as well. We need to ensure that q_c is added only if outlived quorum intersection is preserved. We first present the design intuition, and then an overview of the steps of the protocol.

The goal is to approve adding q_c as a quorum only if it has an outlived intersection with every other quorum q_w in the system. Lemma 8 presents an interesting opportunity to check this condition locally. It states that if an outlived process finds a set self-blocking, then that set has an outlived process. Thus, if we can pass the quorum q_c to an outlived process p_o, and have it check that $q_c \cap q_w$ is self-blocking, then we have that the intersection of q_c and q_w has an outlived process. However, no outlived process is aware of all quorums q_w in



(a) Phase 2: Intersection Check **(b)** Phase 3: Update, Success **(c)** Phase 3: Update, Failure

Figure 8 Phase 2: Intersection Check, and Phase 3: Update

the system. Tentative quorum inclusion can help here. If the outlived process p_o is inside p_w, then by outlived quorum inclusion, p_o has a quorum or tentative quorum q_o (whose well-behaved processes are) included in p_w. The involved quorums are illustrated in Figure 7. If the outlived process p_o check for all its own quorums or tentative quorums q (including q_o), that $q_c \cap q$ is self-blocking then, by Lemma 8, the intersection of q_c and q_o has an outlived process. Since q_o is included in q_w, we get the desire result that the intersection of q_c and q_w has an outlived process. However, how do we reach from the requesting process p to an outlived process p_o in every quorum q_w in the system? Process p that is requesting to add the quorum q_c doesn't know whether it is outlived itself. There is at least a well-behaved process p' in the quorum q_c. By outlived quorum intersection, every quorum q' of p' intersects with every other quorum q_w at an outlived process p_o. Therefore, the protocol takes two hops to reach to the outlived process p_o: process p asks the processes p' of the quorum q_c, and then p' asks the processes p_o in its quorums q'.

Based on the intuition above, we now consider an overview of the protocol. Before adding q_c as a quorum for each process in q_c, the protocol goes through two hops. In the first hop, process p asks each process p' in q_c to perform a check. In the second hop, process p' asks its quorums q' to perform a check. A process p_o in q' checks for every quorum q_o in its set of quorums and tentative quorums that $q_o \cap q_c$ is self-blocking. If the check passes, p_o sends an Ack to p'; otherwise, it sends a Nack. Process p' waits for an Ack from at least one of its quorums q' before sending a commit message back to p. Once p receives a commit message from each process in q_c, it safely adds q_n to its own set of quorums, and requests each process in q_c to add q_c as a quorum.

Protocol Summary. We now present a summary of the protocol. (The details of the protocol are available in Section 14). A process p issues an $Add(q_n)$ request in order to add the quorum q_n to its set of individual minimal quorums, and receives either an *AddComplete* or *AddFail* response. The protocol has three phases: inclusion check, intersection check and update.

Phase 1: Inclusion Check. In phase 1, upon an $Add(q_n)$ request, the requesting process p first checks if quorum inclusion would be preserved for q_n. It sends out $Inclusion(q_n)$ messages to processes in q_n. When a process p' receives the message, it checks whether it already has a quorum which is a subset of q_n, and accordingly sends either *AckInclusion* or *NackInclusion*. Upon receiving these responses, the requesting process p adds the sender p' to the *ack* or *nack* sets respectively. The set *nack* is the set of processes that do not have quorum inclusion. Upon receiving acknowledgment from all processes in q_n, if *nack* is empty, then p simply adds q_n to its set of quorums before issuing the *AddComplete* response. Otherwise, the set $q_c = nack$ is the quorum that should be added to the set of quorums for each process in q_c. To make sure this addition preserves quorum intersection, process p starts phase 2 by sending $CheckAdd(q_c)$ to processes in q_c.

Phase 2: Intersection Check. In phase 2 (Figure 8a), when a $CheckAdd(q_c)$ request is received at a process p', it adds q_c to its tentative set, and sends out a *Check* message to all its quorums. When a process q_o delivers a *Check*, it checks that the intersection of the new quorum q_c with each of its own quorums and tentative quorums is self-blocking. As we saw in the intuition part, this check ensures that there is an outlived process in the intersection. If the checks pass, p_o sends a *CheckAck* message back to p'. Otherwise, it sends a *CheckNack*.

Phase 3: Update. In phase 3 (Figure 8b and Figure 8c), once a process p' receives *CheckAck* messages from one of its quorums, it sends a *Commit* message to the requesting process p. On the other hand, if it receives *CheckNack* from one of its blocking sets, then there is no hope of receiving *CheckAck* from a quorum, and it sends an *Abort* message to p. The requesting process p succeeds if it receives a *Commit* message from every process in the quorum q_c. It fails if it receives an *Abort* message from at least one of them. On the success path, process p adds q_n to its own set of quorums, and sends a *Success* message to processes in q_c before issuing the *AddComplete* response. The *Success* message includes a signature from each process in q_c. On the failure path, process p sends a *Fail* message together with a signature to each process in q_c before issuing an *AddFail* response.

Attack scenarios. Let us consider attack scenarios that motivate the design decisions, and then we get back to the protocol. If the requesting process p is Byzantine, it may send a *Success* message to some processes in q_c, and a *Fail* message to others. Then, a process that receives *Success* adds q_c to its quorums, and another process that receives *Fail* removes q_c from its *tentative* quorums. This would break tentative quorum inclusion. To prevent this, every process p' that receives a *Fail* message echos it to other processes in q_c, and accepts a *Fail* message only when it has received an echo from every process in q_c. Further, a process that accepts a *Success* does not later accept a *Fail* and vice versa. Therefore, we will have the safety invariant that once a process accepts a *Fail*, no other process accepts a *Success*, and vice versa. Another attack scenario is that the requesting process p just sends a *Success* message to some processes in q_c and not others. This attack does not break any of the properties; however, inhibits the progress of other processes. Therefore, every process p' that receives a *Success* message echos it to other processes in q_c.

Success. A process p' accepts a *Success* message to add q_c only if it has not already accepted a *Success* or *Fail* message, and the message comes with valid signatures from all processes in q_c. The signatures are needed to prevent receiving a fake *Success* message from a Byzantine process. Process p' first echos the *Success* message to other processes in q_c. It then adds q_c to its set of quorums, and removes q_c from its tentative quorums.

Failure. A process p' receives a *Fail* message for q_c only if it has not already accepted a *Success* or *Fail* message, and the message comes with a valid signature from the requesting process p. This signature prevents receiving fake *Fail* messages from Byzantine processes. Process p' echos the message, and adds the sender to a set. Once this sender set contains all the processes of q_c, it accepts the *Fail* message, and removes q_c from its tentative quorums.

14.1 Protocol

We present the protocol in three parts: Algorithm 5, Algorithm 6, and Algorithm 7. A well-behaved process p issues an *Add*(q_n) request in order to add the quorum q_n to its set of individual minimal quorums, and receives either an *AddComplete* or *AddFail* response.

Variables and sub-protocols. (1) Each process stores its own set of quorums Q, (2) two sets of processes *ack* and *nack*, (3) a set *tentative* that stores the set of tuples $\langle p, q_c \rangle$ where q_c is a quorum that process p has asked to add, (4) a map *failed* that maps pairs $\langle p, q_c \rangle$ of the requesting process p and the quorum q_c that p wants to add, to the set of processes that a fail message is received from, and (5) a map *succeeded* from the same domain to boolean.

The protocol uses a total-order broadcast *tob*, and authenticated point-to-point links *apl*.

The protocol is executed in three phases: inclusion check, intersection check and update.

Phase 1: Inclusion Check. In phase 1 (Algorithm 5 and Figure 9), upon an $Add(q_n)$ request (at L. 12), the requesting process p first checks if quorum inclusion would be preserved for q_n. It sends out $Inclusion(q_n)$ messages to processes in q_n (at L. 13). When a process p' receives the message (at L. 14), it checks whether it already has a quorum which is a subset of q_n (at L. 15), and accordingly sends either *AckInclusion* or *NackInclusion* (at L. 16 and L. 18). Upon receiving these responses, the requesting process p adds the sender p' to the *ack* or *nack* sets respectively (at L. 20 and L. 22). The set *nack* is the set of processes that do not have quorum inclusion. Upon receiving acknowledgment from all processes in q_n (at L. 23), if *nack* is empty (at L. 24), then q_n is simply added to the set of quorums before issuing the *AddComplete* response. Otherwise, the set *nack* is the quorum q_c that should be added to the set of quorums for each of its members. To make sure this addition preserves quorum intersection, process p starts phase 2 by sending $CheckAdd(nack)$ to processes in *nack* (at L. 28).

Phase 2: Intersection Check. In phase 2 (Algorithm 6 and Figure 8a), when a $CheckAdd(q_c)$ request is received at a process p' (at L. 29), it adds q_c to its tentative set (at L. 30), and it sends out a *Check* message to all its quorums (at L. 31). When a process q_o delivers a *Check* (at L. 32), it checks that the intersection of the new quorum q_c with each of its own quorums in Q and its tentative quorums in *tentative* is **self**-blocking (at L. 33-L. 34). As we saw in the overview part of this section, this check ensures that there is an outlived process in the intersection. Then, the process p_o sends a *CheckAck* message back to p' (at L. 35). Otherwise, it sends a *CheckNack* message (at L. 37).

Phase 3: Update. In phase 3 (Algorithm 7, Figure 8b and Figure 8c), once a process p' receives *CheckAck* messages from one of its quorums (at L. 38), it sends a *Commit* message to the requesting process p (at L. 39). On the other hand, if it receives *CheckNack* from one of its blocking sets (at L. 40), then there is no hope of receiving *CheckAck* from a quorum, and it sends an *Abort* message to the requesting process p (at L. 41). The requesting process p succeeds if it receives a *Commit* message from every process in the quorum q_c. It fails if it receives an *Abort* message from at least one of them. On the success path (at L. 42), process p adds q_n to its own set of quorums (at L. 43), and sends a *Success* message to processes in q_c (at L. 44) before issuing the *AddComplete* response. The *Success* message includes a signature from each process in q_c. On the failure path (at L. 51), process p sends a *Fail* message together with a signature to each process in q_c before issuing an *AddFail* response.

Attack scenarios. Let us consider attack scenarios that motivate the design decisions, and then we get back to the protocol. If the requesting process p is Byzantine, it may send a *Success* message to some processes in q_c, and a *Fail* message to others. Then, a process that receives *Success* adds q_c to its quorums, and another process that receives *Fail* removes q_c

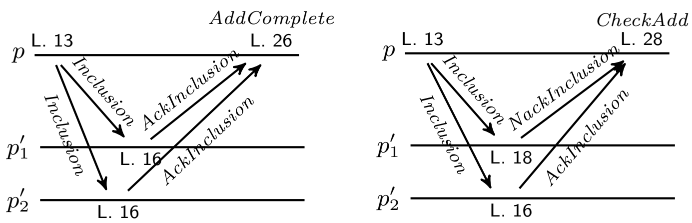


Figure 9 Phase1: Inclusion Check

Algorithm 5 Add quorum (Phase 1: Inclusion check)

Implements: Add
  request : $Add(q_n)$
  response : $AddComplete$ | $AddFail$
Variables:
  Q ▷ The individual minimal quorums of **self**
  $ack, nack : 2^{\mathcal{P}} \leftarrow \emptyset$
  $tentative$: $\mathsf{Set}[\mathcal{P}, 2^{\mathcal{P}}] \leftarrow \emptyset$
  $failed : \langle \mathcal{P}, 2^{\mathcal{P}} \rangle \mapsto 2^{\mathcal{P}} \leftarrow \emptyset$
  $succeeded : \langle \mathcal{P}, 2^{\mathcal{P}} \rangle \mapsto \mathsf{Boolean} \leftarrow \overline{\mathsf{false}}$
Uses:
  $apl : (\cup Q) \cup q_n \mapsto \mathsf{AuthPerfectPointToPointLink}$
upon request $Add(q_n)$
  $apl(p)$ **request** $Inclusion(q_n)$ for each $p \in q_n$
upon response $apl(p)$, $Inclusion(q_n)$
  if $\exists q \in Q.\ q \subseteq q_n$ **then**
    $apl(p)$ **request** $AckInclusion$
  else
    $apl(p)$ **request** $NackInclusion$
upon response $apl(p)$, $AckInclusion$
  $ack \leftarrow ack \cup \{p\}$
upon response $apl(p)$, $NackInclusion$
  $nack \leftarrow nack \cup \{p\}$
upon $ack \cup nack = q_n$
  if $nack = \emptyset$ **then**
    $Q \leftarrow Q \cup \{q_n\}$
    response $AddComplete$
  else
    $apl(p')$ **request** $CheckAdd(nack)$ for each $p' \in nack$

Algorithm 6 Add quorum (Phase 2: Intersection check)

upon response $apl(p)$, $CheckAdd(q_c)$
  $tentative \leftarrow tentative \cup \langle p, q_c \rangle$
  $apl(p_o)$ **request** $Check(\mathbf{self}, q_c)$ for each $p_o \in \cup Q$
upon response $apl(p')$, $Check(p, p', q_c)$
  let $\overline{\langle \_, q'_c \rangle} := tentative$ **in**
  if $\forall q \in \cup\{\overline{q'_c}\} \cup Q.\ q_c \cap q$ *is* **self**-*blocking* **then**
    $apl(p')$ **request** $CheckAck(p, q_c)$
  else
    $apl(p')$ **request** $CheckNack(p, q_c)$

from its *tentative* set. This would break tentative quorum inclusion. To prevent this, every process p' that receives a *Fail* message echos it to other processes in q_c, and processes a *Fail* message only when it has received its echo from every process in q_c. Further, a process that receives a *Success* does not later accept a *Fail*. Therefore, we will have the safety invariant that once a process accepts a *Fail*, no other process accepts a *Success*, and vice versa. Another attack scenario is that the requesting process p just sends a *Success* message to some processes in q_c and not others. This attack does not break any of the properties; however, inhibits the progress of other processes. Therefore, every process p' that receives a *Success* message echos it to other processes in q_c.

Success. A process p' accepts a *Success* message to add q_c (at L. 46) only if the *succeeded* is not set (*i.e.*, p' has not already received a *Success* message, as an optimization), and the message comes with valid signatures from all processes in q_c. The signatures are needed to prevent receiving a fake *Success* message from a Byzantine process. Process p' first echos the *Success* message to other processes in q_c (at L. 48). It then adds q_c to its set of quorums (at L. 49), sets *succeeded* to true (at L. 47), and removes q_c from the *tentative* set (at L. 50).

Failure. A process p' accepts a *Fail* message for q_c (at L. 54) only if *succeeded* is not set (*i.e.*, p' has not received a *Success* message), and the message comes with a valid signature from the requesting process p. This signature prevents receiving fake *Fail* messages from Byzantine processes. Process p' echos the message when it comes from the requesting process p (at L. 55-L. 56), and adds the sender to the set $\mathit{failed}(p, q_c)$ (at L. 57). Once this set contains all the processes of q_c, it removes q_c from its *tentative* set (at L. 58).

14.2 Correctness

We now show that the Add protocol preserves consistency and availability. First we show that although it preserves quorum inclusion only eventually, it does preserve a weak notion of quorum inclusion. We later show that this notion is strong enough to preserve quorum intersection.

Quorum inclusion. In order to preserve quorum inclusion, when q_n is in the system, q_c should be in the system as well. The protocol adds the new quorum q_n for a process p only after q_c is added to the *tentative* set of each process p' in q_c. It then removes q_c from the *tentative* set of p' only after q_c is added to the set of quorums Q of p'. Therefore, when q_n is a quorum of p, q_c is either a quorum or a *tentative* quorum of each process in q_c. We now capture this weak notion as tentative quorum inclusion. As we will see, this weaker notion is sufficient to preserve outlived quorum intersection.

▶ **Definition 35** (Tentative Quorum inclusion)**.** *A quorum system has* tentative *quorum inclusion for P iff for all well-behaved processes p and quorums q of p, if a process p' in q is inside P, then there is a quorum q' such that well-behaved processes of q' are a subset of q, and q' is in either the quorums set Q or the tentative set of p'.*

We now show that the protocol preserves tentative quorum inclusion for the outlived set $\mathcal{O}$, and eventually restores quorum inclusion for $\mathcal{O}$.

▶ **Lemma 36.** *The* Add *protocol preserves tentative quorum inclusion. Further, starting from a quorum system that has quorum inclusion for processes $\mathcal{O}$, it eventually results in a quorum system with quorum inclusion for $\mathcal{O}$.*

Availability. Since the protocol does not remove any quorums, it is straightforward that it preserves availability.

Algorithm 7 Add quorum (Phase 3: Update)

upon response $\overline{\mathit{apl}(p_o), \mathit{CheckAck}(p, q_c)}$ s.t. $\{\overline{p_o}\} \in Q$
  $\mathit{apl}(p)$ **request** $\mathit{Commit}(q_c)^{sig}$
upon response $\overline{\mathit{apl}(p_o), \mathit{CheckNack}(p, q_c)}$ s.t. $\{\overline{p_o}\}$ is **self**-blocking
  $\mathit{apl}(p)$ **request** $\mathit{Abort}(q_c)$
upon response $\overline{\mathit{apl}(p'), \mathit{Commit}(q)^{\sigma}}$ s.t. $\{\overline{p'}\} = q_c \wedge q = q_c$ and $\overline{\sigma \text{ is a valid signature of } p'}$
  $Q \leftarrow Q \cup \{q_n\}$
  $\mathit{apl}(p'')$ **request** $\mathit{Success}(q_c)^{\{\overline{\sigma}\}}$ for each $p'' \in q_c$
  response $\mathit{AddComplete}$
upon response $\mathit{apl}(p)$, $\mathit{Success}(q_c)^{\{\overline{\sigma}\}}$ s.t. $\neg \mathit{succeeded}(p, q_c)$ and $\{\overline{\sigma}\}$ are valid signatures of all processes in q_c
  $\mathit{succeeded}(p, q_c) \leftarrow \mathsf{true}$
  $\mathit{apl}(p'')$ **request** $\mathit{Success}(q_c)^{\{\overline{\sigma}\}}$ for each $p'' \in q_c$
  $Q \leftarrow Q \cup \{q_c\}$
  $\mathit{tentative} \leftarrow \mathit{tentative} \setminus \langle p, q_c \rangle$
upon response $\mathit{apl}(p')$, $\mathit{Abort}(q)$ s.t. $q = q_c \wedge p' \in q_c$
  $\mathit{apl}(p'')$ **request** $\mathit{Fail}(\mathbf{self}, q_c)^{sig}$ for each $p'' \in q_c$
  response $\mathit{AddFail}$
upon response $\mathit{apl}(p^*)$, $\mathit{Fail}(p, q_c)^{\sigma}$ s.t. $\neg \mathit{succeeded}(p, q_c)$ and σ is a valid signature of p
  if $p^* = p$ **then**
    $\mathit{apl}(p'')$ **request** $\mathit{Fail}(p, q_c)^{\sigma}$ for each $p'' \in q_c$
  $\mathit{failed}(p, q_c) \leftarrow \mathit{failed}(p, q_c) \cup \{p^*\}$
  if $q_c \subseteq \mathit{failed}(p, q_c)$ **then**
    $\mathit{tentative} \leftarrow \mathit{tentative} \setminus \langle p, q_c \rangle$

▶ **Lemma 37.** *For every set of processes $\mathcal{O}$, the* Add *protocol preserves quorum availability inside $\mathcal{O}$.*

Quorum Intersection. Now we use the two above properties to show the preservation of quorum intersection.

▶ **Lemma 38.** *If a quorum system has tentative quorum inclusion for processes $\mathcal{O}$, and availability inside $\mathcal{O}$, then the* Add *protocol preserves quorum intersection at $\mathcal{O}$.*

Outlive. Similar to the Leave and Remove protocols, the three above lemmas show that the Add protocol eventually restores an outlived quorum system, while it always maintains outlived quorum intersection.

▶ **Lemma 39** (Preservation of Outlived set)**.** *Starting from a quorum system that is outlived for processes $\mathcal{O}$, the* Add *protocol preserves quorum intersection at $\mathcal{O}$, and eventually results in a quorum system that is outlived for $\mathcal{O}$.*

Immediate from Lemma 38, Lemma 36, and Lemma 37.

▶ **Theorem 40.** *Starting from a quorum system that is outlived for processes $\mathcal{O}$, the* Add *protocol preserves quorum intersection at $\mathcal{O}$, and eventually provides quorum inclusion for $\mathcal{O}$, and availability inside $\mathcal{O}$.*

Remove and Add. Finally, we note that the Leave and Remove protocols (that we saw at Section 6) and the Add protocol can be adapted to execute concurrently. The checks for a blocking set are performed in the Leave and Remove protocols (Algorithm 1), at L. 14 and L. 22, and are performed in the Add protocol (Algorithm 6) at L. 34. The former check considers the *tomb* set and the latter check considers the *pending* set. When the protocols are executed concurrently, both of these sets should be considered. In particular, when the *pending* set is $\{\overline{q_c'}\}$, the check for the former should be that "$\exists q_1, q_2 \in \cup\{\overline{q_c'}\} \cup Q'.\ (q_1 \cap q_2) \setminus (\{p'\} \cup \mathit{tomb})$ is not p'-blocking", and the check for the latter should be that "$\forall q \in \cup\{\overline{q_c'}\} \cup Q.\ q_c \cap q \setminus \mathit{tomb}$ is **self**-blocking".

15 Sink Discovery

Following the graph characterization that we saw in Section 4, we now present a decentralized protocol that can find whether each process is in the sink component of the quorum graph. We first describe the protocol and then its properties.

Protocol. Consider a quorum system with quorum intersection, availability and sharing. The sink discovery protocol in Algorithm 8 finds whether each well-behaved process is in the sink. It also finds the set of its followers. A process p is a follower of process p' iff p has a quorum that includes p'. The protocol has two phases. In the first, it finds the well-behaved minimal quorums, *i.e.*, every minimal quorum that is a subset of well-behaved processes. Since well-behaved minimal quorums are inside the sink, the second phase extends the discovery to other processes in the same strongly connected component.

Variables and sub-protocols. In the quorum system $\mathcal{Q}$, each process **self** stores its own set of individual minimal quorums $Q = \mathcal{Q}(\mathbf{self})$, a map *qmap* from other processes to their quorums which is populated as processes communicate, the *in-sink* boolean that stores whether the process is in the sink, and the set of follower processes F. The protocol uses authenticated point-to-point links *apl*. They provide the following safety and liveness properties. If the sender and receiver are both well-behaved, then the message will be eventually delivered. Every message that a well-behaved process delivers from a well-behaved process is sent by the later, *i.e.*, the identity of a well-behaved process cannot be forged.

Protocol. When processes receive a *Discover* request (at L. 8), they exchange their quorums with each other. In this first phase, each process sends an *Exchange* message with its quorums Q to all processes in its quorums. When a process receives an *Exchange* message (at L. 10), it adds the sender to the follower set F, and stores the received quorums in its *qmap*. As *qmap* is populated, when a process finds that one of its quorums q is a quorum of every other process in q as well (at L. 13), by Lemma 12, it finds that its quorum q is a minimal quorum, and by Theorem 19, it finds itself in the sink. Thus, it sets its *in-sink* variable to `true` in the first phase (at L. 14). The process then sends an *Extend* message with the quorum q to all processes of its own quorums Q (at L. 15). The *Extend* messages are processed in the second phase. The processes of every well-behaved minimal quorum are found in this phase. In Figure 3, since the quorum $\{1, 2\}$ is a quorum for both of 1 and 2, they find themselves in the sink. However, process 3 might receive misleading quorums from process 5, and hence, may not find itself in the sink in this phase.

The processes P_1 of every well-behaved minimal quorum find themselves to be in the sink in the first phase. Let P_2 be the well-behaved processes of the remaining minimal quorums. A pair of minimal quorums have at least a well-behaved process in their intersection. In Figure 3, the two minimal quorums $P_1 = \{1, 2\}$ and $P_2 = \{1, 3, 5\}$ intersect at 1. Therefore, by Lemma 13, every process in P_2 is a neighbor of a process in P_1. Thus, in the second phase, the processes P_1 can send *Extend* messages to processes in P_2, and inform them that they are in the sink. In Figure 3, process 1 can inform process 3. The protocol lets a process accept an *Extend* message containing a quorum q only when the same message comes from the intersection of q and one of its own quorums q' (at L. 16). Let us see why a process in P_2 cannot accept an *Extend* message from a single process. A minimal quorum q that is found in phase 1 can have a Byzantine process p_1. Process p_1 can send an *Extend*(q) message (even with signatures from members of q) to a process p_2 even if p_2 is not a neighbor of p_1, and make p_2 believe that it is in the sink. In Figure 3, the Byzantine process 5 can collect the quorum $\{1, 3, 5\}$ from 1 and 3, and then send an *Extend* message to 4 to make 4 believe that it is inside the sink. Therefore, a process p_2 in P_2 accepts an *Extend*(q) message only when it is received from the intersection of q and one of its own quorums. Since there is a well-behaved process in the intersection of the two quorums, process p_2 can then trust the *Extend* message. When the check passes, p_2 finds itself to be in the sink, and sets the *in-sink* variable to true in the second phase (at L. 17). In Figure 3, when process 3 receives an *Extend* message with quorum $\{1, 2\}$ from 1, since $\{1\}$ is the intersection of the quorum $\{1, 3, 5\}$ of 3, and the received quorum $\{1, 2\}$, process 3 accepts the message.

Algorithm 8 Sink Discovery Protocol

1 **Variables:**
2   Q ▷ The individual minimal quorums of **self**
3   $qmap : \mathcal{P} \mapsto \mathsf{Set}[2^{\mathcal{P}}]$
4   $in\text{-}sink : \mathsf{Boolean} \leftarrow \mathsf{false}$
5   $F : 2^{\mathcal{P}}$
6 **Uses:**
7   $apl : \mathcal{P} \mapsto \mathsf{AuthPerfectPointToPointLink}$
8 **upon request** *Discover*
9   $apl(p)$ **request** $Exchange(Q)$ for each $p \in \cup Q$
10 **upon response** $apl(p)$, $Exchange(Q')$
11   $F \leftarrow F \cup \{p\}$
12   $qmap(p) \leftarrow Q'$
13 **upon** $\exists q \in Q.\ \forall p \in q.\ q \in qmap(p)$
14   $in\text{-}sink \leftarrow \mathsf{true}$
15   $apl(p)$ **request** $Extend(q)$ for each $p \in \cup Q$
16 **upon response** $\overline{apl(p)}, Extend(q)$ s.t. $\exists q' \in Q.\ \{\overline{p}\} = q \cap q'$
17   $in\text{-}sink \leftarrow \mathsf{true}$

Let *ProtoSink* denote the set of well-behaved processes where the protocol sets the *in-sink* variable to true. The discovery protocol is complete: all the well-behaved processes of minimal quorums will eventually know that they are in the sink (*i.e.*, set their *in-sink* to true).

▶ **Lemma 41** (Completeness). *For all $q \in MQ(\mathcal{Q})$, eventually $q \cap \mathcal{W} \subseteq ProtoSink$.*

This result brings an optimization opportunity: the leave and remove protocols can coordinate only when a process inside *ProtoSink* is updated. Although, completeness is sufficient for safety of the optimizations, in the appendix Section 18, we prove both the completeness and accuracy of the two phases in turn. The accuracy property states that *ProtoSink* is a subset of the sink component.

16 AC Leave and Remove Proofs

16.1 Remove, Inclusion-preservation

Lemma 26. *The AC* Leave *and* Remove *protocols preserve active quorum inclusion. Further, starting from a quorum system that has quorum inclusion for processes $\mathcal{O}$, the protocols eventually result in a quorum system with quorum inclusion for $\mathcal{O} \setminus \mathcal{L}$.*

Proof. Consider a quorum system $\mathcal{Q}$ with the set of well-behaved processes $\mathcal{W}$. We assume that $\mathcal{Q}$ has quorum inclusion for a set of processes $\mathcal{O}$. Consider a well-behaved process $p_1 \in \mathcal{W}$ and its quorum $q_1 \in \mathcal{Q}(p_1)$, and a process $p_2 \in q_1 \cap \mathcal{O}$ with a quorum $q_2 \in \mathcal{Q}(p_2)$. For active quorum inclusion, we assume $q_2 \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1 \cap \mathcal{W} \setminus \mathcal{L}$, and show that while a process p is leaving or removing a quorum, this property is preserved. For quorum inclusion, we assume $q_2 \cap \mathcal{W} \subseteq q_1 \cap \mathcal{W}$, and show that this properly will be eventually reconstructed.

Consider a process p that receives the response *LeaveComplete* or *RemoveComplete*. Let $\mathcal{L}' = \mathcal{L} \cup \{p\}$ and $\mathcal{O}' = \mathcal{O} \setminus \mathcal{L}'$. We consider four cases. (1) The requesting process is p_1. We consider two cases. (1.a) If p_1 leaves, then it has no quorums. (1.b) If p_1 removes q_1, no obligation for q_1 remains. (2) If the requesting process is p_2, then $p_2 \notin \mathcal{O}'$ and the property trivially holds. (3) If the requesting process is a process p in q_2 such that $p \neq p_2$, then the two processes p_1 and p_2 will receive *Left* messages (at L. 30) and will eventually remove p from q_1 and q_2 and result in $q_1' = q_1 \setminus \{p\}$ and $q_2' = q_2 \setminus \{p\}$ respectively. To show active quorum inclusion, consider that before the two updates, we have $q_2 \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1 \cap \mathcal{W} \setminus \mathcal{L}$. Depending on the order of the two updates, we should show either $q_2' \cap \mathcal{W} \setminus \mathcal{L}' \subseteq q_1 \cap \mathcal{W} \setminus \mathcal{L}'$ or $q_2 \cap \mathcal{W} \setminus \mathcal{L}' \subseteq q_1' \cap \mathcal{W} \setminus \mathcal{L}'$ for the intermediate states, and both trivially hold. After the two updates, we trivially have $q_2' \cap \mathcal{W}' \setminus \mathcal{L}' \subseteq q_1' \cap \mathcal{W} \setminus \mathcal{L}'$. For eventual quorum inclusion, consider the fact that a message from a well-behaved sender is eventually delivered to a well-behaved receiver. Therefore, the quorums q_1 and q_2 will be eventually updated to the eventual states q_1' and q_2' above. Therefore, if $q_2 \cap \mathcal{W} \subseteq q_1 \cap \mathcal{W}$, then we trivially have $q_2' \cap \mathcal{W} \subseteq q_1' \cap \mathcal{W}$. (4) The requesting process $p \neq p_1$ and is in $q_1 \setminus q_2$. The reasoning is similar to the previous case. ◀

16.2 Remove, Availability-preservation

Lemma 30. *The AC* Leave *and* Remove *protocols preserve active availability. Further, starting from a quorum system that has availability inside processes $\mathcal{O}$, the protocols eventually result in a quorum system with availability inside $\mathcal{O} \setminus \mathcal{L}$.*

Proof. Consider an initial quorum system $\mathcal{Q}$. First, we show that if $\mathcal{Q}$ has active availability inside $\mathcal{O}$, the protocols preserve it. Changes to the quorums of processes in $\mathcal{L}$ does not affect active availability inside $\mathcal{O}$. Other processes can only remove processes in $\mathcal{L}$ from their quorums (at L. 31). Therefore, the inclusion of their quorum inside P modulo $\mathcal{L}$ persists. Second, we show that as processes $\mathcal{L}$ leave or remove quorums, the resulting quorum system $\mathcal{Q}'$ will eventually have availability inside $\mathcal{O} \setminus \mathcal{L}$. Consider a process p that is in $\mathcal{O}$ and not $\mathcal{L}$. We show that there will be a quorum $q' \in \mathcal{Q}'(p)$ such that $q' \subseteq \mathcal{O} \setminus \mathcal{L}$. We have that $\mathcal{Q}$ has availability inside $\mathcal{O}$. Thus, $\mathcal{O}$ are well-behaved, and there is a quorum $q \in \mathcal{Q}(p)$ such that $q \subseteq \mathcal{O}$. Let L be the set of well-behaved processes in $\mathcal{L}$. We show that every p' in L that is in q will be eventually removed from q. Since both p and p' are well-behaved, the *Left* message that p' sends to p (at L. 20 or L. 29) is eventually delivered to p, and p will remove p' from q (at L. 31). Therefore, eventually $q' = q \setminus L$. Thus, since $q \subseteq \mathcal{W}$, $q' = q \setminus \mathcal{L}$. Thus, since $q \subseteq \mathcal{O}$, we have $q' \subseteq \mathcal{O} \setminus \mathcal{L}$

◀

16.3 Remove, Intersection-preservation

Lemma 31. *If a quorum system has quorum intersection at processes $\mathcal{O}$, active availability inside $\mathcal{O}$, and active quorum inclusion for $\mathcal{O}$, then the AC* Leave *and* Remove *protocols preserve quorum intersection at $\mathcal{O} \setminus \mathcal{L}$.*

Proof. Assume that a quorum system $\mathcal{Q}$ has quorum intersection at processes $\mathcal{O}$, availability inside $\mathcal{O}$ and active quorum inclusion for $\mathcal{O}$. We have two cases for the requesting process **self**: it is either in $\mathcal{O}$ or not. In the latter case, it cannot affect the assumed intersection at $\mathcal{O}$.

Now let us consider the case where the leaving process **self** is in $\mathcal{O}$. Consider two well-behaved processes p_1 and p_2 with quorums $q_1 \in \mathcal{Q}(p_1)$ and $q_2 \in \mathcal{Q}(p_2)$. Let I be the intersection of q_1 and q_2 in $\mathcal{O}$, *i.e.*, $I = q_1 \cap q_2 \cap \mathcal{O}$. Assume that **self** is in the intersection of q_1 and q_2, *i.e.*, $\textbf{self} \in I$. We assume that **self** receives a *LeaveComplete* or *RemoveComplete* response. We show that the intersection of the two quorums has another process in $\mathcal{O}$. Let L be the subset of processes in I that have received a *LeaveComplete* or or *RemoveComplete* response before **self** receives hers. After the processes $\mathcal{L}$ and **self** receive a response, the quorums will incrementally shrink (at L. 31) where the final smallest quorums are $q_1' = q_1 \setminus (\mathcal{L} \cup \{\textbf{self}\})$ and $q_2' = q_2 \setminus (\mathcal{L} \cup \{\textbf{self}\})$ respectively. Let $\mathcal{L}' = \mathcal{L} \cup \{\textbf{self}\}$ and $\mathcal{O}' = \mathcal{O} \setminus \mathcal{L}'$. We show that even the smallest quorums have intersection in $\mathcal{O}'$, *i.e.*, $q_1' \cap q_2' \cap \mathcal{O}' \neq \emptyset$.

A *LeaveComplete* or *RemoveComplete* response is issued (at L. 28) when processing a *Check* request. The total-order broadcast *tob* totally orders the *Check* deliveries. Let p^* be the process in $(L \cup \{\textbf{self}\})$ that is ordered last in the total order. By Lemma 26, active quorum inclusion is preserved. Therefore, since p^* is in q_1 and $\mathcal{O}$, there is a quorum $q_1^* \in \mathcal{Q}(p^*)$ such that $q_1^* \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1 \cap \mathcal{W} \setminus \mathcal{L}$. Similarly, we have that there is a quorum $q_2^* \in \mathcal{Q}(p^*)$ such that $q_2^* \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_2 \cap \mathcal{W} \setminus \mathcal{L}$. Since p^* has received a complete response, $q_1^* \cap q_2^* \setminus (\{p^*\} \cup \mathit{tomb})$ is p^*-blocking (by the condition at L. 22). The total-order broadcast *tob* ordered the *Check* deliveries for every process in the set $L \cup \{\textbf{self}\} \setminus \{p^*\}$ before the *Check* delivery for p^*. Therefore, since every process that gets a complete response is added to the *tomb* set (at L. 26), the *tomb* set of p^* includes these processes, *i.e.*, $L \cup \{\textbf{self}\} \setminus \{p^*\} \subseteq \mathit{tomb}$. Therefore, by substitution of *tomb*, we have $q_1^* \cap q_2^* \setminus (L \cup \{\textbf{self}\})$ is a blocking set for p^*. By the definition of L above, we have that the processes $\mathcal{L} \setminus L$ are not in the intersection of q_1, q_2 and $\mathcal{O}$. Therefore, $q_1^* \cap q_2^* \setminus (\mathcal{L} \cup \{\textbf{self}\})$ is a blocking set for p^*. Therefore, $q_1^* \cap q_2^* \setminus \{\textbf{self}\}$ is an active blocking set for p^*. By Lemma 29, there is a process $p \in \mathcal{O} \setminus \mathcal{L}$ such that $p \in q_1^* \cap q_2^* \setminus (\{\textbf{self}\})$. We have $(q_1^* \cap q_2^* \setminus (\{\textbf{self}\})) \cap (\mathcal{O} \setminus \mathcal{L}) \neq \emptyset$. Distribution of $\setminus$ give us $(q_1^* \setminus \mathcal{L} \cup \{\textbf{self}\}) \cap (q_2^* \setminus \mathcal{L} \cup \{\textbf{self}\}) \cap (\mathcal{O} \setminus \mathcal{L} \cup \{\textbf{self}\}) \neq \emptyset$. By active quorum inclusion, we have $q_1^* \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_1 \cap \mathcal{W} \setminus \mathcal{L}$ and $q_2^* \cap \mathcal{W} \setminus \mathcal{L} \subseteq q_2 \cap \mathcal{W} \setminus \mathcal{L}$. Thus, we have $(q_1 \setminus \mathcal{L} \cup \{\textbf{self}\}) \cap (q_2 \setminus \mathcal{L} \cup \{\textbf{self}\}) \cap (\mathcal{O} \setminus \mathcal{L} \cup \{\textbf{self}\}) \neq \emptyset$. Thus, $q_1' \cap q_2' \cap \mathcal{O}' \neq \emptyset$.

◀

17 Add Proofs

17.1 Add, Inclusion-preservation

Lemma 36. *The* Add *protocol preserves tentative quorum inclusion. Further, starting from a quorum system that has quorum inclusion for processes $\mathcal{O}$, it eventually results in a quorum system with quorum inclusion for $\mathcal{O}$.*

Proof. Consider a quorum system $\mathcal{Q}$ with the set of well-behaved processes $\mathcal{W}$. Consider a set of well-behaved processes $\mathcal{O}$, a well-behaved process $p_1 \in \mathcal{W}$ and its quorum $q_1 \in \mathcal{Q}(p_1)$, and a process $p_2 \in q_1 \cap \mathcal{O}$ with a quorum $q_2 \in \mathcal{Q}(p_2)$. For tentative quorum inclusion, we assume $q_2 \cap \mathcal{W} \subseteq q_1 \cap \mathcal{W}$, and that q_2 is in either $\mathcal{Q}(p_2)$ or the *tentative* set of p_2, and show that while a process p is adding a quorum, this property is preserved.

We consider a case for each such line where a quorum is added.

Case (1): The quorum q_n is added at L. 25. By L. 21-L. 22, and L. 24, the process p has received the *AckInclusion* message from every process $p' \in q_n$. Since before sending *AckInclusion*, a well-behaved p' makes sure it has a quorum inside q_n (at L. 15) and $\mathcal{O} \subseteq \mathcal{W}$, we have $\forall p' \in q_n \cap \mathcal{O}.\ \exists q' \in \mathcal{Q}(p').\ q' \subseteq q_n$. This is sufficient for quorum inclusion that is stronger than tentative quorum inclusion.

Case (2): The quorum q_n is added at L. 43: The sets $q_n \setminus q_c$ and q_c are the processes in q_n which, respectively, do and do not satisfy quorum inclusion with their existing quorums (L. 15-18 and L. 21-22). Further, $q_c \subseteq q_n$. Therefore, we only need to show tentative quorum inclusion for processes p' in q_c. Before adding q_n, the process p receives the *Commit* message from every process $p' \in q_c$ (at L. 42). Before sending the *Commit* message, every well-behaved process $p' \in q_c$ receives *CheckAck* messages from all the processes of one of its quorums including itself (at L. 38). p_o only send *CheckAck* message after receives *Check* message from p' at L. 31, which is after q_c has been added to *tentative* of p' Therefore, (a) before q_n is added as a quorum of p, every well-behaved process p' in q_c adds q_c to its *tentative* set. We will show below that (b) every well-behaved process p' in q_c, removes q_c from the *tentative* set only after it is already added to its set of quorums. Since $\mathcal{O} \subseteq \mathcal{W}$, (a) and (b) above show tentative quorum inclusion for $\mathcal{O}$.

We now show the assertion (b) above. p' removes q_c from its *tentative* set only after it receives *Success* or *Fail* messages. We consider a case for each Case (2.1): q_c is removed from *tentative* at L. 50. q_c is removed from *tentative* only after q_c is added to the set of quorums Q. Case (2.2): q_c is removed at L. 59, which is after process p' receiving *Fail* messages from every member of q_c (at L. 58) and verifying the signature from p (at L. 54). However, the process p has sent a *Success* message (at L. 44) after q_n is added. If a well-behaved process sends a *Success* message, then it returns the response *AddComplete* and the add process finishes. Therefore, since p is a well-behaved process, it does not send a *Fail* message and this case does not happen.

Case (3): The quorum q_c is added at L. 49. This add happens only after the current process p' receives a *Success* message (at L. 46) . Upon delivery of a *Success* message, p' validates a signature from every process in q_c for *Commit*. Therefore, by an argument similar to Case (2), we have that (a) before a process adds q_c to its quorums, every well-behaved process in q_c has added q_c to its *tentative* set. We will show below that (b) every well-behaved process in q_c removes q_c from the *tentative* set only after it is already added to its set of quorums. Since $\mathcal{O} \subseteq \mathcal{W}$, (a) and (b) above show tentative quorum inclusion for $\mathcal{O}$.

We now show the assertion (b) above. A process p'' removes q_c from its *tentative* set only after delivery of *Success* or *Fail* messages and we consider a case for each. Case (3.1): q_c is removed at L. 50. The argument is similar to Case (2.1). Case (3.2): q_c is removed at L. 59, which is after receiving the *Fail* messages from all the members of q_c including p'. However, before q_c is added to the quorums of p', p' set *succeeded* to true at L. 47. The condition $\neg succeeded(p, q_c)$ at L. 54 prevents it from sending *Fail* after receiving *Success*. Therefore, since p' has received a *Success* message, then it does not send a *Fail* message. Therefore, p'' can not remove q_c from its *tentative* at L. 59.

For eventual quorum inclusion, consider the requesting process p that adds q_n to its quorums. The argument is similar to the Case (1) and Case (2) above for tentative quorum inclusion. In the second case, before adding q_n (at L. 43), the process p sends the *Success* message to processes in q_c (at L. 44). Therefore, all the well-behaved processes in q_c eventually add q_c to their set of quorums (at L. 49). Therefore, q_c and q_n will eventually have quorum inclusion. ◀

17.2 Add, Availability-preservation

Lemma 37. *For every set of processes $\mathcal{O}$, the* Add *protocol preserves quorum availability inside $\mathcal{O}$.*

Proof. The Add protocol does not explicitly remove any quorums. An implicit removal can happen when a process has a quorum that is a subset of another. We show that if the addition of quorum q_c leads to an implicit removal of a quorum, availability is preserved. If q_c is a subset of an existing quorum, then even if either of the quorums is removed, availability is preserved. Further, q_c is not a superset of any quorum q of a process p in q_c. Otherwise, q is a subset of q_n and the process p would not be included in q_c. ◀

17.3 Add, Intersection-preservation

Lemma 38. *If a quorum system has tentative quorum inclusion for processes $\mathcal{O}$, and availability inside $\mathcal{O}$, then the* Add *protocol preserves quorum intersection at $\mathcal{O}$.*

Proof. Consider well behaved processes $\mathcal{O}$, and a quorum system $\mathcal{Q}$ with tentative quorum inclusion for $\mathcal{O}$, availability inside $\mathcal{O}$, and quorum intersection at $\mathcal{O}$. Consider a well-behaved process p that requests $Add(q_n)$. The new quorums that are added to the quorum system are q_c and its superset q_n. Consider a well-behaved process p_w with a quorum q_w that is either an existing quorum or a tentative quorum that passed the all the checks. We should show that both q_c and q_n intersect q_w at $\mathcal{O}$.

We assumed that there is a well-behaved process p' in q_n. We consider two cases: Case (1) The well-behaved process p' is in q_c. A process adds q_c (at L. 49) only after receiving a *Success* message (at L. 46). The process p sends a *Success* message (at L. 44) only after receiving the *Commit* message form every process in q_c (at L. 42). The well-behaved process p' sends a *Commit* message (at L. 39) only after receiving a *CheckAck* message from one of its quorums q' (at L. 38). By quorum intersection at $\mathcal{O}$, the intersection of q' and q_w has a process p_o in $\mathcal{O}$. By tentative quorum inclusion for $\mathcal{O}$, there is a quorum q_o of p_o such that $q_o \cap \mathcal{W} \subseteq q_w \cap \mathcal{W}$ and either q_o is a quorum of p_o or a member of its *tentative* set. Since p_o has sent an *CheckAck* message (at L. 35), it has passed the check that $q_o \cap q_c$ is p_o-blocking (at L. 33-L. 34). Since $\mathcal{O}$ is available and p_o is in $\mathcal{O}$, then by Lemma 8, we have that $q_o \cap q_c \cap \mathcal{O} \neq \emptyset$. Since $q_o \cap \mathcal{W} \subseteq q_w \cap \mathcal{W}$, and $\mathcal{O} \subseteq \mathcal{W}$, then $q_w \cap q_c \cap \mathcal{O} \neq \emptyset$. Since $q_c \subseteq q_n$, we have $q_w \cap q_n \cap \mathcal{O} \neq \emptyset$.

Case (2) The well-behaved process p' is in $q_n \setminus q_c$. A process in q_n is not in q_c only if it already satisfies quorum inclusion. (L. 15-18 and L. 21-22). Therefore, p' has a quorum q such that $q \subseteq q_n$. Thus, the quorum intersection for q implies quorum intersection for q_n. If q_c has a well-behaved process, the proof follows the previous case. Otherwise, it has no well-behaved process, and quorum intersection is only required for well-behaved processes. ◀

18 Sink Discovery Proofs

18.1 Sink Discovery, Completeness

Let the set *ProtoSink* partition into $ProtoSink_1$ and $ProtoSink_2$ that denote the set of well-behaved processes at which the protocol sets the *in-sink* variable to true in phase 1 (at L. 14), and in phase 2 (at L. 17) respectively.

Well-behaved minimal quorums will eventually be in $ProtoSink_1$.

▶ **Lemma 42** (Completeness for Phase 1). *Forall $q \in MQ(\mathcal{Q})$, if $q \subseteq \mathcal{W}$, eventually $q \subseteq ProtoSink_1$.*

Proof. Consider a well-behaved minimal quorum q. By Lemma 12, every process in q is in a quorum of every other process in q. Since all processes in q are well-behaved, they send out *Exchange* messages to all the other processes in q (at L. 9). Since these processes are well-behaved, they will eventually receive each other's messages, and record each other quorums (at L. 10). Therefore, each of them will eventually satisfy the condition, and set *in-sink* to true (at L. 14-L. 13). ◀

Well-behaved processes in the minimal quorums will eventually be in either $ProtoSink_1$ or $ProtoSink_2$.

▶ **Lemma 43** (Completeness for Phase 2). *Forall $q \in MQ(\mathcal{Q})$, eventually $q \cap \mathcal{W} \subseteq ProtoSink_1$ or $q \cap \mathcal{W} \subseteq ProtoSink_2$.*

Proof. Consider a minimal quorum q which is not found in phase 1, *i.e.*, it is not added to $ProtoSink_1$. Since the quorum system is available (for a set of processes), there exists a well-behaved quorum q'. By the consistency property, q and q' have an intersection $\{\overline{p}\}$. Since q' is well-behaved, $\{\overline{p}\}$ are well-behaved. Since each process p in $\{\overline{p}\}$ is in q', by Lemma 42, p sets its *in-sink* variable to true (at L. 14), and then sends out *Extend* messages to its neighbors: all processes of its individual minimal quorums (at L. 15). Since each process p is in q, by Lemma 13, every well-behaved process of q is a neighbor of p. Therefore, all well-behaved processes in q will receive the *Extend* messages from the intersection $\{\overline{p}\}$, and set their *in-sink* variable to true. ◀

18.2 Sink Discovery, Accuracy

We saw above that completeness is sufficient for safety of the optimizations. We now consider the accuracy property: every well-behaved process that sets its *in-sink* variable to true is in the sink component. Let us consider an attack scenario for accuracy. A group of Byzantine processes fake to be a quorum and send an *Extend* message to make a process that it is outside the sink believe that it is in the sink. This can violate accuracy. Therefore, a check *validq* is needed (at L. 16) to ensure that the processes in q are a valid quorum in the system. For example, the heterogeneous quorum systems of both Stellar [43] and Ripple [58] use hierarchies of processes and size thresholds to recognize quorums. We prove the accuracy of the two phases in turn that results in the following lemma. Let $Sink(\mathcal{Q})$ denote processes in the sink component of $\mathcal{Q}$.

▶ **Lemma 44** (Accuracy). *$ProtoSink \subseteq Sink(\mathcal{Q})$.*

▶ **Lemma 45** (Accuracy of Phase 1). *$ProtoSink_1 \subseteq Sink(\mathcal{Q})$*

Proof. Processes in $\mathit{ProtoSink}_1$ set *in-sink* to true (at L. 14) after they check (at L. 13) that one of their quorums q in an individual minimal quorum of all its members. By Lemma 12, q is a minimal quorum, and by Theorem 19, q is in the sink. ◀

▶ **Lemma 46** (Accuracy for Phase 2). $\mathit{ProtoSink}_2 \subseteq \mathit{Sink}(\mathcal{Q})$

Proof. Consider a process p^* in $\mathit{ProtoSink}_2$. It sets *in-sink* to true (at L. 17) only after receiving an *Extend* message containing a quorum q from a set of processes $P' = \{\overline{p'}\}$ such that P' is the intersection of q and a quorum of p^* (at L. 16). Further, the *validq* check ensures that q has at least one well-behaved process p_w (at L. 16). Since well-behaved processes only send *Extend* messages with a minimal quorum, q is a minimal quorum. Since p^* receives a message from p_w, there is an edge from p_w to p^* in the quorum graph. We show that there is a path from p^* to p_w. By Lemma 14, there are edges from p^* to all members of at least one minimal quorum q'. By consistency, there is at least one well-behaved process p'_w in the intersection of q' and q. There is an edge from p^* to p'_w. By Lemma 15, there is an edge from p'_w to p_w. Thus, there is a path from p^* to p_w though p'_w. Therefore, they are strongly connected. By Lemma 45, p_w is in the sink, Therefore, p^* is in the sink as well. ◀