

Tight Bounds on Channel Reliability via Generalized Quorum Systems

Alejandro Naser-Pastoriza
IMDEA Software Institute
Universidad Politécnica de Madrid
Madrid, Spain

Alexey Gotsman
IMDEA Software Institute
Madrid, Spain

Gregory Chockler
University of Surrey
Guildford, United Kingdom

Fedor Ryabinin
IMDEA Software Institute
Universidad Politécnica de Madrid
Madrid, Spain

Abstract

Communication channel failures are a major concern for the developers of modern fault-tolerant systems. However, while tight bounds for process failures are well-established, extending them to include channel failures has remained an open problem. We introduce *generalized quorum systems* – a framework that characterizes the necessary and sufficient conditions for implementing atomic registers, atomic snapshots, lattice agreement and consensus under arbitrary patterns of process-channel failures. Generalized quorum systems relax the connectivity constraints of classical quorum systems: instead of requiring bidirectional reachability for every pair of write and read quorums, they only require some write quorum to be *unidirectionally* reachable from some read quorum. This weak connectivity makes implementing registers particularly challenging, because it precludes the traditional request/response pattern of quorum access, making classical solutions like ABD inapplicable. To address this, we introduce novel logical clocks that allow write and read quorums to reliably track state updates without relying on bidirectional connectivity.

CCS Concepts

• Theory of computation → Distributed algorithms.

Keywords

Distributed algorithms, lower bounds, atomic registers, atomic snapshots, lattice agreement, consensus

ACM Reference Format:

Alejandro Naser-Pastoriza, Gregory Chockler, Alexey Gotsman, and Fedor Ryabinin. 2025. Tight Bounds on Channel Reliability via Generalized Quorum Systems. In *ACM Symposium on Principles of Distributed Computing (PODC '25)*, June 16–20, 2025, Huatulco, Mexico. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3732772.3733529>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PODC '25, Huatulco, Mexico

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1885-4/25/06
<https://doi.org/10.1145/3732772.3733529>

1 Introduction

Tolerating communication channel failures is one of the toughest challenges facing the developers of modern distributed systems. In fact, according to a recent study [8], a majority of failures attributed to network partitions led to catastrophic effects whose resolution often required redesigning core system mechanisms. What makes channel failures even harder to deal with is that they need to be tolerated in conjunction with ordinary process failures. The resulting vast space of faulty behaviors makes the analysis of computability questions under these failure conditions particularly challenging.

It is therefore not surprising that, with a few exceptions, prior work has studied process and channel failures in isolation from each other. In particular, it is well-known that tolerating up to k process crashes in a fully connected network of n processes is possible for a range of problems (e.g., registers and consensus) if and only if $n \geq 2k + 1$ [9, 19]. Subsequent work [27, 31] generalized this result to the case when the set of possible faulty behaviors is specified as a *fail-prone system* – a collection of *failure patterns*, each defining a set of processes that can crash in a single execution [33]. In this case, the registers and consensus are implementable under a given fail-prone system if there exists a *read-write quorum system (QS)* in which: any read and write quorums intersect (Consistency); and some read and write quorums of correct processes are available in every execution (Availability). In this paper we extend these results to failure patterns that may include arbitrary combinations of both process and channel failures – namely, process crashes and channel disconnections.

EXAMPLE 1. Consider a set of processes $\mathcal{P} = \{a, b, c, d\}$. In Figure 1 we depict a fail-prone system \mathcal{F} , consisting of failure patterns f_i , $i = 1..4$ (ignore the sets R_i and W_i for now). Under failure pattern f_1 , processes a, b, c are correct, while d may crash. Channels (c, a) , (a, b) , (b, a) are correct, while all others may disconnect.

A plausible conjecture for a tight bound on connectivity under this kind of a fail-prone system would require the existence of a read-write quorum system QS^+ that preserves Consistency but modifies Availability to require that the processes within the available read or write quorum are strongly connected by correct channels. This ensures that some process can communicate with both a read and a write quorum (e.g., one in their intersection), directly enabling the execution of algorithms like ABD [9] and Paxos [30]. In fact, QS^+ was shown to be sufficient for consensus [4, 17, 18, 23], and

more recently, for registers and consensus under a more aggressive message loss model [36]. The latter work also proved the existence of QS^+ to be necessary in the special case of $n = 2k + 1$.

These results, however, leave a noticeable gap. While the existence of QS^+ is optimal for $n = 2k + 1$, it is unknown whether it is necessary for arbitrary fail-prone systems, such as those with $k < \lfloor \frac{n-1}{2} \rfloor$, or those not based on failure thresholds at all [33, 34]. Surprisingly, we answer this question in the negative. We show that atomic registers, atomic snapshots, (single-shot) lattice agreement and (partially synchronous) consensus can be implemented even when none of the available read quorums is strongly connected by correct channels. Instead, we only require enough connectivity for some strongly connected write quorum to be *unidirectionally* reachable from some read quorum – a condition that we formalize via a novel *generalized quorum system* (GQS, §3). We prove that, given a fail-prone system \mathcal{F} comprising an arbitrary set of process-channel failure patterns, the above problems are implementable under \mathcal{F} if and only if \mathcal{F} admits a GQS.

EXAMPLE 2. Consider the fail-prone system $\mathcal{F} = \{f_i \mid i = 1..4\}$ in Figure 1. The families of read quorums $\mathcal{R} = \{R_i \mid i = 1..4\}$ and write quorums $\mathcal{W} = \{W_i \mid i = 1..4\}$ form a generalized quorum system: each write quorum W_i is strongly connected and is reachable from the read quorum R_i through channels correct under the failure pattern f_i . None of the read quorums R_i is strongly connected, thus relaxing the connectivity requirements of QS^+ .

Our results also show that any solution to the above-mentioned problems can guarantee termination only within the write quorums of some GQS (e.g., processes a and b under the failure pattern f_1 from Figure 1). Such a restricted termination guarantee is expected in our setting, because channel failures may isolate some correct processes, making it impossible to ensure termination at all of them. Accordingly, to prove our upper bounds, we first present an implementation of atomic registers on top of a GQS that ensures wait-freedom within its write quorums (§5). Since atomic snapshots can be constructed from atomic registers [2], and lattice agreement from snapshots [11], the upper bounds for snapshots and lattice agreement follow. To prove the lower bounds, we go in the reverse direction: we first prove the bound for lattice agreement (§6); then the above-mentioned constructions imply the bounds for snapshots and registers.

Implementing registers on top of a GQS presents a unique challenge. To complete a register operation invoked at a process, this process needs to communicate with both a read and a write quorum – a typical pattern in algorithms like ABD. However, the limited connectivity within read quorums means that the process cannot query a read quorum by simply sending messages to its members and awaiting responses.

EXAMPLE 3. Assume that a register operation is invoked at process a under failure pattern f_1 from Figure 1. In this case all channels coming into process $c \in R_1$ may have disconnected. This makes it impossible for a to request information from this member of the read quorum by sending a message to it. Of course, c could periodically push information to a through the correct channel (c, a) , without waiting for explicit requests. But because the network is asynchronous, it is challenging for a to determine whether

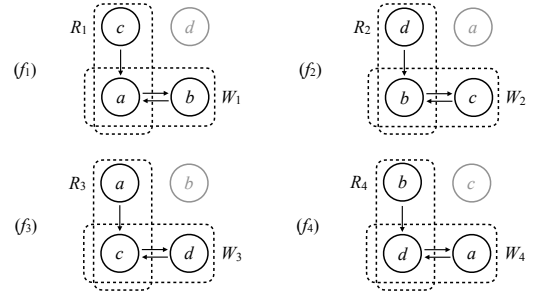


Figure 1: A fail-prone system $\mathcal{F} = \{f_i \mid i = 1..4\}$ and a generalized quorum system $(\mathcal{F}, \mathcal{R}, \mathcal{W})$, for $\mathcal{R} = \{R_i \mid i = 1..4\}$ and $\mathcal{W} = \{W_i \mid i = 1..4\}$. Solid circles denote correct processes; gray circles, processes that may crash; solid arrows, reliable channels; missing arrows, channels that may disconnect.

the information it receives from c is up to date, i.e., captures all updates preceding the current operation invocation at a – a critical requirement for ensuring linearizability.

We address this challenge using novel logical clocks that processes use to tag the information they push downstream. These clocks are cooperatively maintained by processes when updating a write quorum or querying a read quorum. We encapsulate the corresponding protocol into reusable *quorum access functions*, which are then used to construct an ABD-like algorithm for registers.

Finally, we also show that the existence of a GQS is a tight bound on the connectivity required for implementing consensus under partial synchrony (§7). Interestingly, implementing consensus under arbitrary process and channel failures is simpler than implementing registers, because a process can exploit the eventual timeliness of the network to determine if the information it receives is up to date.

2 System Model and Preliminary Definitions

System model. We consider an *asynchronous* system with a set \mathcal{P} of n processes that may fail by crashing. A process is *correct* if it never crashes, and *faulty* otherwise. Processes communicate by exchanging messages through a set of unidirectional channels \mathcal{C} : for every pair of processes $p, q \in \mathcal{P}$ there is a channel $(p, q) \in \mathcal{C}$ that allows p to send messages to q . Channels can be *correct* or *faulty*. A *correct* channel is reliable: it delivers all messages sent by a correct process. A *faulty* channel fails by disconnection: from some point on it drops all messages sent through it.

Fail-prone systems. To state our results, we need a way of specifying which processes and channels can fail during an execution. Following our previous work [36], we do this by generalizing the classical notion of a fail-prone system [33] to include channel failures. A *failure pattern* is a pair $(P, C) \in 2^{\mathcal{P}} \times 2^{\mathcal{C}}$ that defines which processes and channels are allowed to fail in a single execution. We assume that \mathcal{C} contains only channels between correct processes, as channels incident to faulty processes are considered faulty by default: $(p, q) \in \mathcal{C} \implies \{p, q\} \cap P = \emptyset$. Given a failure pattern $f = (P, C)$, an execution σ of the system is *f-compliant* if at most the processes in P and at most the channels in C fail in σ . A *fail-prone system* \mathcal{F} is a set of failure patterns (see Figure 1).

EXAMPLE 4. The standard failure model where any minority of processes may fail and channels between correct processes are reliable is captured by the fail-prone system $\mathcal{F}_M = \{(Q, \emptyset) \mid Q \subseteq \mathcal{P} \wedge |Q| \leq \lfloor \frac{n-1}{2} \rfloor\}$.

Safety. We consider algorithms that implement the following objects in the model just introduced: multi-writer multi-reader (MWMR) atomic registers, single-writer multi-reader (SWMR) atomic snapshots, (single-shot) lattice agreement [2] and consensus. These objects provide *operations* that can be invoked by clients – e.g., reads and writes in the case of registers. Their safety properties are standard. We introduce them for some of the objects as needed and defer the rest to [37, §A].

Liveness. Defining liveness properties under our failure model is subtle, because channel failures may isolate some correct processes, making it impossible to ensure termination at all of them. To deal with this, we adapt the classical notions of obstruction-freedom and wait-freedom by parameterizing them based on the failures allowed and the subsets of correct processes where termination is required. We use the weaker obstruction-freedom in our lower bounds and the stronger wait-freedom in our upper bounds.

For a failure pattern $f = (P, C)$ and a set of processes $T \subseteq \mathcal{P} \setminus P$, we say that an algorithm \mathcal{A} is (f, T) -wait-free if, for every process $p \in T$, operation op , and f -compliant fair¹ execution σ of \mathcal{A} , if op is invoked by p in σ , then op eventually returns. For example, we may require an algorithm to be (f_1, T_1) -wait-free for the failure pattern f_1 in Figure 1 and $T_1 = \{a, b\}$. This means that operations invoked at a and b must return despite the failures of process d and channels (a, c) , (b, c) and (c, b) .

An operation op eventually executes solo in an execution σ if there exists a suffix σ' of σ such that all operations concurrent with op in σ' are invoked by faulty processes. We say that an algorithm \mathcal{A} is (f, T) -obstruction-free if, for every process $p \in T$, operation op , and f -compliant fair execution σ of \mathcal{A} , if op is invoked by p and eventually executes solo in σ , then op eventually returns. This notion of obstruction-freedom aligns with its well-known shared memory counterparts [10, 21, 25].

We lift the notions of obstruction-freedom and wait-freedom to a fail-prone system \mathcal{F} and a *termination mapping* $\tau : \mathcal{F} \rightarrow 2^{\mathcal{P}}$ – a function mapping each failure pattern $f \in \mathcal{F}$ to a subset of correct processes whose operations are required to terminate. We say that an algorithm \mathcal{A} is (\mathcal{F}, τ) -wait-free if, for every $f \in \mathcal{F}$, \mathcal{A} satisfies $(f, \tau(f))$ -wait-freedom. We define (\mathcal{F}, τ) -obstruction-freedom similarly.

EXAMPLE 5. The standard guarantee of wait-freedom under a minority of process failures corresponds to (\mathcal{F}_M, τ_M) -wait-freedom, where \mathcal{F}_M is defined in Example 4 and τ_M selects the set of all correct processes: $\forall f = (Q, \emptyset) \in \mathcal{F}_M. \tau_M(f) = \mathcal{P} \setminus Q$.

3 Generalized Quorum Systems

Fault-tolerant distributed algorithms commonly ensure the consistency of replicated state using *quorums*, i.e., intersecting sets of processes. It is common to separate quorums into two classes – read and write quorums – so that the intersection is required only

between a pair of quorums from different classes. For example, in a variant of the ABD register implementation values are stored at a write quorum and fetched from a read quorum [9, 31]. The intersection between read and write quorums ensures that a read operation observes the latest completed write. Similarly, in the Paxos consensus algorithm decision proposals are stored at a phase-2 quorum (analogous to a write quorum) and information about previously accepted proposals is gathered from a phase-1 quorum (analogous to a read quorum) [27, 30]. The intersection between phase-1 and phase-2 quorums ensures the uniqueness of decisions. The classical definition of a quorum system considers only process failures, not channel failures [33, 34]. In our framework, we can express this definition as follows.

DEFINITION 1. Consider a fail-prone system \mathcal{F} that disallows channel failures between correct processes: $\forall (P, C) \in \mathcal{F}. C = \emptyset$. A **quorum system** is a triple $(\mathcal{F}, \mathcal{R}, \mathcal{W})$, where $\mathcal{R} \subseteq 2^{\mathcal{P}}$ is a family of read quorums, $\mathcal{W} \subseteq 2^{\mathcal{P}}$ is a family of write quorums, and the following conditions hold:

- Consistency. For all $R \in \mathcal{R}$ and $W \in \mathcal{W}, R \cap W \neq \emptyset$.
- Availability. For all $f \in \mathcal{F}$, there exist $R \in \mathcal{R}$ and $W \in \mathcal{W}$ such that all processes in $R \cup W$ are correct according to f .

EXAMPLE 6. Consider a system with n processes, where at most $k \leq \lfloor \frac{n-1}{2} \rfloor$ processes can fail. The following triple $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ is a quorum system [27]:

- The fail-prone system is such that at most k processes can fail, and channels between correct processes do not fail:
 $\mathcal{F} = \{(P, \emptyset) \mid P \subseteq \mathcal{P} \wedge |P| \leq k\}$.
- Read quorums are of size at least $n - k$:
 $\mathcal{R} = \{R \mid R \subseteq \mathcal{P} \wedge |R| \geq n - k\}$.
- Write quorums are of size at least $k + 1$:
 $\mathcal{W} = \{W \mid W \subseteq \mathcal{P} \wedge |W| \geq k + 1\}$.

The above quorum system illustrates the usefulness of distinguishing between read and write quorums: this allows trading off smaller write quorums for larger read quorums. In the special case where $k = \lfloor \frac{n-1}{2} \rfloor$ we get $\mathcal{R} = \mathcal{W}$, so that both read and write quorums are majorities of processes.

The existence of a classical quorum system is sufficient to implement atomic registers, atomic snapshots, lattice agreement and consensus in a model without channel failures. We now generalize the notion of a quorum system to accommodate such failures. A straightforward generalization would preserve Consistency, but modify Availability to require that the processes within the available read or write quorum are strongly connected by correct channels. This ensures that some process can communicate with both a read and a write quorum (e.g., one in their intersection), directly enabling the execution of algorithms like ABD. Surprisingly, we find that this strong connectivity requirement is unnecessarily restrictive: the above-listed problems can be solved even when read quorums are not strongly connected. To define the corresponding notion of a quorum system, we use the following concepts:

- *Network graph*: let $\mathcal{G} = (P, C)$ be the directed graph with all processes as vertices and all channels as edges.
- *Residual graph*: for a failure pattern $f = (P, C)$, let $\mathcal{G} \setminus f$ be the subgraph of \mathcal{G} obtained by removing all processes in P , their incident channels, and all channels in C .

¹Recall that an execution σ is *fair* if every process that is correct in σ takes an infinite number of steps in σ .

- *f-availability*: a set $Q \subseteq \mathcal{P}$ is *f-available* if it contains only processes correct according to f and it is strongly connected in $\mathcal{G} \setminus f$. This implies that all processes in Q can communicate with each other via channels correct according to f .
- *f-reachability*: a set $W \subseteq \mathcal{P}$ is *f-reachable* from a set $R \subseteq \mathcal{P}$ if both W and R contain only processes correct according to f , and every member of W can be reached by every member of R via a directed path in $\mathcal{G} \setminus f$.

EXAMPLE 7. In Figure 1, for each $i = 1..4$, W_i is f_i -available, and W_i is f_i -reachable from R_i .

DEFINITION 2. A **generalized quorum system** is a triple $(\mathcal{F}, \mathcal{R}, \mathcal{W})$, where \mathcal{F} is a fail-prone system, $\mathcal{R} \subseteq 2^{\mathcal{P}}$ is a family of read quorums, $\mathcal{W} \subseteq 2^{\mathcal{P}}$ is a family of write quorums, and the following conditions hold:

- Consistency. For every $R \in \mathcal{R}$ and $W \in \mathcal{W}$, $R \cap W \neq \emptyset$.
- Availability. For all $f \in \mathcal{F}$, there exist $W \in \mathcal{W}$ and $R \in \mathcal{R}$ such that W is *f-available*, and W is *f-reachable* from R .

Informally speaking, Availability guarantees that an operational write quorum can unidirectionally receive information from an operational read quorum under any failure scenario defined by \mathcal{F} .

A classical quorum system is a special case of a generalized quorum system. Indeed, if \mathcal{F} disallows channel failures between correct processes, then every correct write quorum can be trivially reached from every read quorum, and Definition 2 becomes equivalent to Definition 1.

EXAMPLE 8. Consider the fail-prone system $\mathcal{F} = \{f_i \mid i = 1..4\}$ in Figure 1 and let $\mathcal{W} = \{W_i \mid i = 1..4\}$ and $\mathcal{R} = \{R_i \mid i = 1..4\}$. Then the triple $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ is a generalized quorum system. Indeed:

- Consistency. For each $i, j = 1..4$, $R_i \cap W_j \neq \emptyset$.
- Availability. For each $i = 1..4$, W_i is f_i -available, and W_i is f_i -reachable from R_i .

Note that the above $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ is a valid generalized quorum system even though the processes in each read quorum are not strongly connected via correct channels: some pair of processes are only connected unidirectionally. This relaxation allows read quorums to be formed under a broader range of failure scenarios. In contrast, processes within each write quorum validating Availability must be strongly connected via correct channels. In fact, for a given failure pattern f we can show that different write quorums validating Availability with respect to f must also be strongly connected via correct channels.

PROPOSITION 1. Let $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ be a generalized quorum system. For each $f \in \mathcal{F}$, the following set of processes is strongly connected in $\mathcal{G} \setminus f$:

$$U = \bigcup \{W \mid W \in \mathcal{W} \wedge (W \text{ is } f\text{-available}) \wedge \exists R \in \mathcal{R}. (W \text{ is } f\text{-reachable from } R)\}.$$

PROOF. Fix $f \in \mathcal{F}$ and let $p_1, p_2 \in U$. Then there exist $W_1, W_2 \in \mathcal{W}$ such that

- $(p_1 \in W_1) \wedge (W_1 \text{ is } f\text{-available}) \wedge \exists R_1 \in \mathcal{R}. (W_1 \text{ is } f\text{-reachable from } R_1)$; and
- $(p_2 \in W_2) \wedge (W_2 \text{ is } f\text{-available}) \wedge \exists R_2 \in \mathcal{R}. (W_2 \text{ is } f\text{-reachable from } R_2)$.

Because any read and write quorums intersect, $W_1 \cap R_2 \neq \emptyset$. Fix $q \in W_1 \cap R_2$. Since W_1 is *f-available*, there exists a path via correct channels from p_1 to q . And since W_2 is *f-reachable* from R_2 , there exists a path via correct channels from q to p_2 . Thus, p_2 is reachable from p_1 via correct channels. We can analogously show that p_1 is reachable from p_2 via correct channels, as required. \square

For $f \in \mathcal{F}$ we denote the strongly connected component of $\mathcal{G} \setminus f$ containing U by U_f : this component includes all write quorums that validate Availability with respect to f . The Availability property of the generalized quorum system ensures that $U_f \neq \emptyset$.

4 Main Results

We now state our main results for asynchrony, proved in §5-6: the existence of a generalized quorum system is a tight bound on the process and channel failures that can be tolerated by any implementation of MWMMR atomic registers, SWMMR atomic snapshots, and lattice agreement (we handle partially synchronous consensus in §7). The following theorem establishes that the existence of a generalized quorum system is sufficient to implement each of the three objects. For each $f \in \mathcal{F}$, these implementations provide wait-freedom within the strongly connected component U_f .

THEOREM 1. Let $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ be a generalized quorum system and $\tau : \mathcal{F} \rightarrow 2^{\mathcal{P}}$ be the termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) = U_f$. Then there exists an (\mathcal{F}, τ) -wait-free implementation for each of the following objects: MWMMR atomic registers, SWMMR atomic snapshots, and lattice agreement.

The next theorem establishes a matching lower bound, showing that the existence of a generalized quorum system is necessary to implement any of the three objects. This assumes a weak termination guarantee that only requires obstruction-freedom to hold at some non-empty set of processes for each failure pattern.

THEOREM 2. Let \mathcal{F} be a fail-prone system and $\tau : \mathcal{F} \rightarrow 2^{\mathcal{P}}$ be a termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) \neq \emptyset$. Assume that there exists an (\mathcal{F}, τ) -obstruction-free implementation of any of the following objects: MWMMR atomic registers, SWMMR atomic snapshots, or lattice agreement. Then there exist \mathcal{R} and \mathcal{W} such that $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ is a generalized quorum system. Moreover, for each $f \in \mathcal{F}$, we have $\tau(f) \subseteq U_f$.

The theorem also shows U_f is the largest set of processes for which termination can be guaranteed under the failure pattern f . Furthermore, the two theorems imply that if termination can be guaranteed for at least one process, then it can also be guaranteed for all processes in U_f .

EXAMPLE 9. Consider the generalized quorum system $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ from Figure 1. Then $U_{f_1} = \{a, b\}$, $U_{f_2} = \{b, c\}$, $U_{f_3} = \{c, d\}$ and $U_{f_4} = \{d, a\}$. Consider τ such that $\tau(f_i) = U_{f_i}$, $i = 1..4$. By Theorem 1, there exists an (\mathcal{F}, τ) -wait-free implementation for any of the three objects considered. Suppose now that we change f_1 to f'_1 that additionally fails the channel (a, b) . Let $\mathcal{F}' = \{f'_1, f_2, f_3, f_4\}$. It is easy to check that there do not exist \mathcal{R}' and \mathcal{W}' that would form a generalized quorum system $(\mathcal{F}', \mathcal{R}', \mathcal{W}')$. Thus, Theorem 2 implies that there is no implementation of any of the three objects that would provide obstruction-freedom (and, by extension, wait-freedom) anywhere under \mathcal{F}' .

To prove the upper bound (Theorem 1), we construct an (\mathcal{F}, τ) -wait-free implementation of MWMR atomic registers (§5). Since SWMR atomic snapshots can be constructed from the registers [2], and lattice agreement can in turn be constructed from snapshots [11], we naturally get the upper bounds for the latter two problems. We consider the weakest variant of lattice agreement [11], which is single-shot and therefore cannot be used for implementing multi-shot objects, such as registers. Thus, to prove the lower bound (Theorem 2), we first establish it for lattice agreement (§6); then the above-mentioned constructions imply the lower bound for snapshots and registers.

5 Upper Bound for Atomic Registers

Fix a generalized quorum system $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ and a termination mapping $\tau : \mathcal{F} \rightarrow 2^{\mathcal{P}}$ such that $\tau(f) = U_f$ for each $f \in \mathcal{F}$. Without loss of generality, we assume that the connectivity relation of the graph $\mathcal{G} \setminus f$ is transitive for each $f \in \mathcal{F}$: if not, transitivity can be easily simulated by having all processes forward every received message. To implement atomic registers using the quorum system, we need to come up with a way for a process to contact a read and a write quorum, as required by algorithms such as ABD [9]. This is challenging in our setting because processes within a read quorum may not be strongly connected by correct channels.

EXAMPLE 10. Consider the generalized quorum system in Figure 1 and the failure pattern f_1 . The available read and write quorums are $R_1 = \{a, c\}$ and $W_1 = \{a, b\}$. Since Theorem 1 requires $\tau(f_1) = U_{f_1}$, a register implementation validating it has to ensure wait-freedom within W_1 . However, all channels coming into c may have failed. This makes it impossible for a member of W_1 , such as a , to request information from some of the members of R_1 , such as c , by sending them an explicit message to this end.

Quorum access functions. We encapsulate the mechanics necessary to deal with this challenge using the following *quorum access functions*, which allow a process to obtain up-to-date information from a quorum. We assume that the top-level protocol, such as a register implementation, maintains a state from a set \mathcal{S} at each process. Then the interface of the access functions is as follows:

- $\text{quorum_get}() \in 2^{\mathcal{S}}$: returns the states of all members of some read quorum; and
- $\text{quorum_set}(u)$: applies the *update function* $u : \mathcal{S} \rightarrow \mathcal{S}$ to the states of all members of some write quorum.

We require that these functions satisfy the following properties:

- **Validity.** For any state s returned by $\text{quorum_get}()$, there exists a subset of previous invocations $\{\text{quorum_set}(u_i) \mid i = 1..k\}$ such that s is the result of applying the update functions in $\{u_i \mid i = 1..k\}$ to the initial state in some order.
- **Real-time ordering.** If $\text{quorum_set}(u)$ terminates, then its effect is visible to any later $\text{quorum_get}()$, i.e., the set returned by $\text{quorum_get}()$ includes at least one state to which u has been applied.
- **Liveness.** The functions are (\mathcal{F}, τ) -wait-free, i.e., they terminate at every member of $\tau(f)$ for any $f \in \mathcal{F}$.

As we show in the following, quorum access functions are sufficient to program an ABD-like algorithm for atomic registers.

```

1 state  $\in \mathcal{S}$  // opaque state of the top-level protocol
2 seq  $\leftarrow 0$ 
3 function quorum_get():
4   seq  $\leftarrow$  seq + 1
5   send GET_REQ(seq) to all
6   wait until received {GET_RESP(seq,  $s_j$ ) |  $p_j \in R$ }
7   return  $\{s_j \mid p_j \in R\}$ 
8 when received GET_REQ( $k$ ) from  $p_j$ 
9   send GET_RESP( $k$ , state) to  $p_j$ 
10 function quorum_set( $u$ ):
11  seq  $\leftarrow$  seq + 1
12  send SET_REQ(seq,  $u$ ) to all
13  wait until received {SET_RESP(seq) |  $p_j \in W$ }
14  from some  $W \in \mathcal{W}$ 
15 when received SET_REQ( $k$ ,  $u$ ) from  $p_j$ 
16  state  $\leftarrow u(\text{state})$ 
17  send SET_RESP( $k$ ) to  $p_j$ 

```

Figure 2: Quorum access functions for a classical quorum system: the protocol at a process p_i .

Quorum access functions for classical quorum systems. To illustrate the concept of quorum access functions, in Figure 2 we provide their implementation using a classical quorum system that disallows channel failures (Definition 1). Each process stores the state of the top-level protocol, such as a register implementation, in a state variable. This state is managed by the implementation of the quorum access functions, but its structure is opaque to this implementation: it can only manipulate the state by applying update functions passed by the callers. Each process also maintains a monotonically increasing seq number (initially 0), which is used to generate a unique identifier for each quorum access function invocation at this process. This identifier is then added to every message exchanged by the implementation, so that the process can tell which messages correspond to which invocations.

Upon a call to $\text{quorum_get}()$ at a process, it broadcasts a GET_REQ message (line 5). Any process receiving this responds with a GET_RESP message that carries its current state (line 9). The $\text{quorum_get}()$ invocation returns when it accumulates such responses from a read quorum (line 6). Upon a call to $\text{quorum_set}(u)$ at a process, it broadcasts a SET_REQ message carrying the function u (line 12). Any process receiving this applies u to its current state and responds with SET_RESP (line 16). The $\text{quorum_set}()$ invocation returns when it accumulates such responses from a write quorum (line 13).

It is easy to see that the above implementation ensures the Validity and Real-time ordering properties, the latter because any read and write quorums intersect. Finally, this implementation guarantees wait-freedom at every correct process (Liveness): Availability ensures that there exist a read quorum and a write quorum of correct processes; then the fact that the fail-prone system disallows channel failures ensures that any process can communicate with these quorums.

Quorum access functions for generalized quorum systems. We now show how we can implement the quorum access functions for generalized quorum systems, despite the lack of strong connectivity within read quorums. We use Example 10 for illustration. Recall that in this example we need to ensure termination within W_1 and, in particular, at a . An implementation of `quorum_get()` at a needs to obtain state snapshots from every member of R_1 . However, channel failures may make it impossible for a to send a message to $c \in R_1 \setminus W_1$ to request this information. Of course, c could just periodically propagate its state to all processes it is connected to, without them asking for it explicitly: by Availability, W_1 is f -reachable from R_1 , so that messages sent by c will eventually reach a . But because the network is asynchronous, *the process a cannot easily determine when the information it receives is up to date*, i.e., when it captures the effects of all `quorum_set()` invocations that completed before `quorum_get()` was called at a – as necessary to satisfy Real-time ordering.

To address this challenge, each process maintains a monotonically increasing logical clock, stored in the variable `clock` (initially 0; this clock is different from the usual logical clocks for tracking causality [22, 29]). We then modify the implementation of the quorum access functions in Figure 2 as shown in Figure 3:

- **Periodic state propagation (line 12):** A process periodically advances its clock and propagates its current state and clock in a `GET_RESP` message, without waiting for an explicit request. The clock value in the message indicates the logical time by which the process had this state.
- **Clock updates during state changes (line 21):** When handling a `SET_REQ(k, u)` message, a process increments its clock and sends it as part of the `SET_RESP` message. The process thereby indicates the logical time by which it has incorporated u into its state.
- **Delaying the completion of `quorum_set` (line 15):** Upon a `quorum_set(u)` invocation, the process broadcasts u in a `SET_REQ` message and waits for `SET_RESP` messages from every member of some write quorum W_{set} , as in the classical implementation. The process selects the highest clock value among the responses received and stores it in a variable c_{set} . It then waits until some read quorum R_{set} reports having clock $\geq c_{\text{set}}$ before completing the invocation. As we show in the following, this wait serves to ensure that future `quorum_get()` invocations will observe the update u .
- **Clock cutoff for `quorum_get` (line 3):** Upon a `quorum_get()` invocation, the process first determines a clock value c_{get} , delimiting how up-to-date the states it will return should be. To this end, the process broadcasts a `CLOCK_REQ` message. Any process receiving this message responds with a `CLOCK_RESP` message that carries its current clock value. The process executing `quorum_get()` waits until it receives such responses from all members of some write quorum and picks the highest clock value among those received – this is the desired clock cut-off c_{get} . Finally, the process waits until it receives `GET_RESP(s_j, c_j)` messages with $c_j \geq c_{\text{get}}$ from all members of some read quorum and returns the collected states s_j to the caller.

```

1 state  $\in \mathcal{S}$  // opaque state of the top-level protocol
2 seq, clock  $\leftarrow 0, 0$ 
3 function quorum_get():
4   seq  $\leftarrow$  seq + 1
5   send CLOCK_REQ(seq) to all
6   wait until received {CLOCK_RESP(seq,  $c_j$ ) |  $p_j \in W_{\text{get}}$ }
7   from some  $W_{\text{get}} \in \mathcal{W}$ 
8    $c_{\text{get}} \leftarrow \max\{c_j \mid p_j \in W_{\text{get}}\}$ 
9   wait until received {GET_RESP( $s_j, c_j$ ) |  $p_j \in R_{\text{get}}$ }
10  from some  $R_{\text{get}} \in \mathcal{R}$  where  $\forall j. c_j \geq c_{\text{get}}$ 
11  return  $\{s_j \mid p_j \in R_{\text{get}}\}$ 
12 when received CLOCK_REQ( $k$ ) from  $p_j$ 
13   send CLOCK_RESP( $k, \text{clock}$ ) to  $p_j$ 
14 periodically
15   clock  $\leftarrow$  clock + 1
16   send GET_RESP(state, clock) to all
17 function quorum_set( $u$ ):
18   seq  $\leftarrow$  seq + 1
19   send SET_REQ(seq,  $u$ ) to all
20   wait until received {SET_RESP(seq,  $c_j$ ) |  $p_j \in W_{\text{set}}$ }
21   from some  $W_{\text{set}} \in \mathcal{W}$ 
22    $c_{\text{set}} \leftarrow \max\{c_j \mid p_j \in W_{\text{set}}\}$ 
23   wait until received {GET_RESP( $_, c_j$ ) |  $p_j \in R_{\text{set}}$ }
24   from some  $R_{\text{set}} \in \mathcal{R}$  where  $\forall j. c_j \geq c_{\text{set}}$ 
25 when received SET_REQ( $k, u$ ) from  $p_j$ 
26   state  $\leftarrow u(\text{state})$ 
27   clock  $\leftarrow$  clock + 1
28   send SET_RESP( $k, \text{clock}$ ) to  $p_j$ 

```

Figure 3: Quorum access functions for a generalized quorum system: the protocol at a process p_i .

Note that `quorum_set` and `quorum_get` operations work in tandem: `quorum_set` delays its completion until clocks have advanced sufficiently at a read quorum; this allows `quorum_get` to establish a clock cutoff capturing all prior completed updates. Interestingly, `quorum_set` uses read quorums for this purpose, while `quorum_get` uses write quorums – an inversion of the traditional quorum roles.

It is easy to see that this implementation validates the Validity property of the quorum access functions. We now prove that it also validates Real-time ordering. First, consider c_{set} computed by a `quorum_set(u)` invocation (lines 16-19). The following lemma shows that querying the states of a read quorum with clocks $\geq c_{\text{set}}$ is sufficient to observe u .

LEMMA 1. *Assume that c_{set} is computed by a `quorum_set(u)` invocation at a process p_{set} (lines 16-19). Consider a set of messages $\{\text{GET_RESP}(s_j, c_j) \mid p_j \in R\}$ sent by all members of some read quorum R , each with $c_j \geq c_{\text{set}}$. Then some s_j has incorporated u .*

PROOF. To compute the value of c_{set} , the process p_{set} waits for `SET_RESP($_, c_j$)` messages from all members p_j of a write quorum W_{set} . Since any read quorum intersects any write quorum, there exists $p_l \in R \cap W_{\text{set}}$. Because $p_l \in W_{\text{set}}$, this process sends a

SET_RESP($_, c'_l$) message to p_{set} during the $\text{quorum_set}(u)$ invocation. The process p_{set} computes $c_{\text{set}} = \max\{c'_j \mid p_j \in W_{\text{set}}\}$, so that $c_{\text{set}} \geq c'_l$. Because $p_l \in R$, this process also sends a GET_RESP(s_l, c_l) message with $c_l \geq c_{\text{set}} \geq c'_l$. At this moment p_l has clock = c_l . Hence, if p_l sends the GET_RESP(s_l, c_l) message before sending the SET_RESP($_, c'_l$) message, then due to the increment at line 23 and the fact that process clocks never decrease, we must have $c_l < c'_l$. But this contradicts the fact $c_l \geq c'_l$ that we established earlier. Hence, p_l must send the SET_RESP($_, c'_l$) message before sending the GET_RESP(s_l, c_l) message and, therefore, s_l incorporates u . \square

In the light of the above lemma, for $\text{quorum_get}()$ to validate Real-time ordering, it just needs to find a clock value that is $\geq c_{\text{set}}$ of any previously completed $\text{quorum_set}()$. As we show in the following proof, this is precisely what is achieved by the computation of c_{get} in $\text{quorum_get}()$.

THEOREM 3 (Real-time ordering). *If a $\text{quorum_set}(u)$ operation terminates, then its effect is visible to any subsequent $\text{quorum_get}()$ invocation.*

PROOF. Assume that $\text{quorum_set}(u)$ terminates at a process p_{set} before $\text{quorum_get}()$ is invoked at a process p_{get} . Since any read quorum intersects any write quorum, there exists $p_l \in R_{\text{set}} \cap W_{\text{get}}$. Since $p_l \in R_{\text{set}}$, before $\text{quorum_set}(u)$ terminated, p_{set} received GET_RESP($_, c'_l$) from p_l with $c'_l \geq c_{\text{set}}$ (line 20). Hence, by the time $\text{quorum_set}(u)$ terminated, p_l had clock $\geq c_{\text{set}}$. We also have $p_l \in W_{\text{get}}$, and thus p_{get} received CLOCK_RESP($_, c_l$) from p_l at line 6. This message was sent at line 11 after $\text{quorum_get}()$ had been invoked, and thus, after $\text{quorum_set}(u)$ had terminated. Above we established that by the latter point p_l had clock $\geq c_{\text{set}}$, and thus, $c_l \geq c_{\text{set}}$. The process p_{get} computed $c_{\text{get}} = \max\{c_j \mid p_j \in W_{\text{get}}\}$ at line 7, so that $c_{\text{get}} \geq c_l \geq c_{\text{set}}$. Then each GET_RESP(s_j, c_j) that p_{get} received from $p_j \in R_{\text{get}}$ at line 8 satisfies $c_j \geq c_{\text{get}} \geq c_{\text{set}}$. By Lemma 1, some s_j has incorporated u , as required. \square

THEOREM 4 (Liveness). *The protocol in Figure 3 is (\mathcal{F}, τ) -wait-free.*

PROOF. We prove the liveness of $\text{quorum_get}()$; the case of $\text{quorum_set}()$ is analogous. Fix a failure pattern $f \in \mathcal{F}$ and a process $p \in \tau(f)$ that executes $\text{quorum_get}()$. By Availability, there exist $W \in \mathcal{W}$ and $R \in \mathcal{R}$ such that W is f -available, and W is f -reachable from R . By Proposition 1, $W \subseteq U_f$. Then since $\tau(f) = U_f$ and $p \in \tau(f)$, p is strongly connected to W via channels correct under f . Therefore, p will be able to exchange the CLOCK_REQ and CLOCK_RESP messages with every member of W , thus exiting the wait at line 6. Recall that W is f -reachable from R and each process periodically increments its clock value and propagates it in a GET_RESP message (line 12). Hence, p will receive GET_RESP messages from all members of R with high enough clock values to exit the wait at line 8 and return. \square

Registers via quorum access functions. In Figure 4 we give an implementation of an atomic register using the above quorum access functions, which validates Theorem 1. The implementation follows the structure of a multi-writer/multi-reader variant of ABD [9, 31]: the main novelty of our protocol lies in the implementation of the quorum access functions.

```

1  $\mathcal{S} = \text{Value} \times \text{Version}$  // register state type
2 function write( $x$ ):
3    $S \leftarrow \text{quorum\_get}()$ 
4    $(k, \_) \leftarrow \max\{s.\text{ver} \mid s \in S\}$ 
5    $t \leftarrow (k + 1, i)$ 
6    $u \leftarrow (\lambda s. \text{if } t > s.\text{ver} \text{ then return } (x, t) \text{ else return } s)$ 
7    $\text{quorum\_set}(u)$ 
8 function read():
9    $S \leftarrow \text{quorum\_get}()$ 
10  let  $s' \in S$  be such that  $\forall s \in S. s'.\text{ver} \geq s.\text{ver}$ 
11   $u \leftarrow (\lambda s. \text{if } s'.\text{ver} > s.\text{ver} \text{ then return } s' \text{ else return } s)$ 
12   $\text{quorum\_set}(u)$ 
13  return  $s'.$ val

```

Figure 4: The atomic register protocol at a process p_i .

Let Value be the domain of values the register stores. Like ABD, our implementation tags values with versions from $\text{Version} = \mathbb{N} \times \mathbb{N}$, ordered lexicographically. A version is a pair of a monotonically increasing number and a process identifier. Each register process maintains a state consisting of a pair (val, ver), where val is the most recent value written to the register at this process and ver is its version. Hence, we instantiate \mathcal{S} in Figure 3 to $\mathcal{S} = \text{Value} \times \text{Version}$, with (0, 0) as the initial state. To execute a write(x) operation (line 2 in Figure 4), a process proceeds in two phases:

- **Get phase.** The process uses $\text{quorum_get}()$ to collect the states from some read quorum. Based on these, it computes a unique version t , higher than every received one.
- **Set phase.** The process next uses $\text{quorum_set}()$ to store the pair (x, t) at some write quorum. To this end, it passes as an argument a function u that describes how each member of the write quorum should update its state (expressed using λ -notation, line 6). Given a state s of a write-quorum member, the function acts as follows: if the new version t is higher than the old version $s.\text{ver}$, then the function returns a state with the new value x and version t ; otherwise it returns the unchanged state s . Recall that the implementation of $\text{quorum_set}()$ uses the result of this function to replace the states at the members of a write quorum.

To execute a read() operation (line 8), a process follows two similar phases:

- **Get phase.** The process uses $\text{quorum_get}()$ to collect the states from all members of some read quorum. It then picks the state s' with the largest version among those received. The value part $s'.\text{val}$ of this state will be returned as a response to the read().
- **Set phase.** Before returning from read(), the process must guarantee that the value read will be seen by any subsequent operation. To this end, it writes the state s' back using similar steps to the Set phase for the write() operation.

Liveness of the quorum access functions trivially implies that the protocol in Figure 4 is (\mathcal{F}, τ) -wait-free. The Real-time ordering property can be used to show that the protocol is linearizable [26]; we defer the proof to [37, §B].

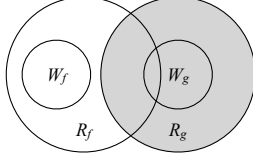


Figure 5: Illustration of the sets W_k and R_k for $k \in \{f, g\}$.

6 Lower Bound for Lattice Agreement

In this section we prove Theorem 2 for lattice agreement. Then the existing constructions of lattice agreement from snapshots and registers [2, 11] imply that the theorem holds also for the latter objects. Recall that in the lattice agreement problem, each process p_i may invoke an operation $\text{propose}(x_i)$ with its input value x_i . This invocation terminates with an output value y_i . Both input and output values are elements of a semi-lattice with a partial order \leq and a join operation \sqcup . An algorithm that solves lattice agreement must satisfy the following conditions for all i and j :

- Comparability. Either $y_i \leq y_j$ or $y_j \leq y_i$.
- Downward validity. If process p_i outputs y_i , then $x_i \leq y_i$.
- Upward validity. If process p_i outputs y_i , then $y_i \leq \sqcup X$, where X is the set of x_j for which $\text{propose}(x_j)$ was invoked.

We rely on the following lemma, which establishes that processes where obstruction-freeness holds must be strongly connected by correct channels. We defer its proof to [37, §C].

LEMMA 2. *Let f be a failure pattern and $T \subseteq \mathcal{P}$. If some algorithm \mathcal{A} is an (f, T) -obstruction-free implementation of lattice agreement, then T is strongly connected in $\mathcal{G} \setminus f$.*

PROOF OF THEOREM 2 FOR LATTICE AGREEMENT. Let \mathcal{A} be an (\mathcal{F}, τ) -obstruction-free implementation of lattice agreement. For a failure pattern $f \in \mathcal{F}$, Lemma 2 implies that the set $\tau(f)$ of processes where obstruction-freeness holds must be transitively connected via correct channels. Let then W_f be the strongly connected component of $\mathcal{G} \setminus f$ containing $\tau(f)$ and R_f be the set of processes that can reach W_f in $\mathcal{G} \setminus f$, including W_f itself. Finally, let $\mathcal{R} = \{R_f \mid f \in \mathcal{F}\}$ and $\mathcal{W} = \{W_f \mid f \in \mathcal{F}\}$.

We show that $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ is a generalized quorum system. Assume by contradiction that this is not the case. Note that for every $f \in \mathcal{F}$ we have that W_f is f -available and W_f is f -reachable from R_f . Thus, $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ satisfies Availability. Since by assumption $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ is not a generalized quorum system, then it must fail to satisfy Consistency. Hence, there exist $f, g \in \mathcal{F}$ such that $W_f \cap W_g = \emptyset$. Refer to Figure 5 for a visual depiction.

CLAIM 1. *For $k \in \{f, g\}$, R_k is unreachable from $\mathcal{P} \setminus R_k$ in $\mathcal{G} \setminus k$.*

CLAIM 2. *For $k \in \{f, g\}$, $R_k \setminus W_k$ is unreachable from W_k in $\mathcal{G} \setminus k$.*

Let \mathcal{L} be a semi-lattice with partial order \leq such that $x_1, x_2 \in \mathcal{L}$, $x_1 \not\leq x_2$ and $x_2 \not\leq x_1$. Let α_1 be a fair execution of \mathcal{A} where the processes and channels in f fail at the beginning, a process $p_1 \in \tau(f)$ invokes $\text{propose}(x_1)$, and no other operation is invoked in α_1 . Because $p_1 \in \tau(f)$ and \mathcal{A} is $(f, \tau(f))$ -obstruction-free, the $\text{propose}(x_1)$ operation must eventually terminate with a return value y_1 . By Downward validity, $x_1 \leq y_1$. By Upward validity, since

no other $\text{propose}()$ was invoked, $y_1 \leq x_1$. Thus, $y_1 = x_1$. Let α_2 be the prefix of α_1 ending with the response to $\text{propose}()$. By Claim 1, R_f is unreachable from $\mathcal{P} \setminus R_f$, and thus, the actions by processes in R_f do not depend on those by processes in $\mathcal{P} \setminus R_f$. Then $\alpha_3 = \alpha_2|_{R_f}$, the projection of α_2 to processes in R_f , is an execution of \mathcal{A} . By Claim 2, $R_f \setminus W_f$ is unreachable from W_f , and thus, the actions by processes in $R_f \setminus W_f$ do not depend on those by processes in W_f . Then $\alpha = \alpha_3|_{R_f \setminus W_f} \alpha_3|_{W_f}$ is an execution of \mathcal{A} . Finally, $\alpha_3|_{R_f \setminus W_f}$ contains no propose invocations (only startup steps).

Let β_1 be a fair execution of \mathcal{A} that starts with all the actions from $\alpha_3|_{R_f \setminus W_f}$, followed by the failure of all processes and channels in g , followed by a $\text{propose}(x_2)$ invocation by a process $p_2 \in \tau(g)$. Because $p_2 \in \tau(g)$ and \mathcal{A} is $(g, \tau(g))$ -obstruction-free, the $\text{propose}(x_2)$ operation must eventually terminate with a return value y_2 . By Downward validity, $x_2 \leq y_2$. By Upward validity, since no other $\text{propose}()$ was invoked, $y_2 \leq x_2$. Thus, $y_2 = x_2$. Let β_2 be the prefix of β_1 ending with the response to $\text{propose}()$ and let δ be the suffix of β_2 such that $\beta_2 = \alpha_3|_{R_f \setminus W_f} \delta$. By Claim 1, R_g is unreachable from $\mathcal{P} \setminus R_g$, and thus, the actions in δ by processes in R_g do not depend on those by processes in $\mathcal{P} \setminus R_g$. Then, $\beta = \alpha_3|_{R_f \setminus W_f} \delta|_{R_g}$ is an execution of \mathcal{A} .

Consider the execution $\sigma = \alpha_3|_{R_f \setminus W_f} \alpha_3|_{W_f} \delta|_{R_g}$ where no process or channel fails. We have:

$$\begin{aligned} \sigma|_{R_f \setminus R_g} &= (\alpha_3|_{R_f \setminus W_f} \alpha_3|_{W_f} \delta|_{R_g})|_{R_f \setminus R_g} = \\ &(\alpha_3|_{R_f \setminus W_f} \alpha_3|_{W_f})|_{R_f \setminus R_g} = \alpha|_{R_f \setminus R_g}. \end{aligned}$$

Thus, σ is indistinguishable from α to the processes in $R_f \setminus R_g$. Also, because $W_f \cap R_g = \emptyset$, we have:

$$\sigma|_{R_g} = (\alpha_3|_{R_f \setminus W_f} \alpha_3|_{W_f} \delta|_{R_g})|_{R_g} = (\alpha_3|_{R_f \setminus W_f} \delta|_{R_g})|_{R_g} = \beta|_{R_g}.$$

Thus, σ is indistinguishable from β to the processes in R_g . Finally, $\sigma|_{\mathcal{P} \setminus (R_f \cup R_g)} = \varepsilon$. Thus, for every process, σ is indistinguishable to this process from some execution of \mathcal{A} . Furthermore, each message received by a process in σ has previously been sent by another process. Therefore, σ is an execution of \mathcal{A} . However, in this execution p_1 decides x_1 , p_2 decides x_2 , and x_1, x_2 are incomparable in \mathcal{L} . This contradicts the Comparability property of lattice agreement. The contradiction derives from assuming that $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ is not a generalized quorum system, so the required follows. Finally, for each $f \in \mathcal{F}$ we have $\tau(f) \subseteq W_f \subseteq U_f$. \square

7 Tight Bound for Consensus

We now move from the asynchronous model we have used so far to the *partially synchronous model* [19]. We show that, in this model, the existence of a generalized quorum system is also a tight bound on the process and channel failures that can be tolerated by any implementation of consensus. The partially synchronous model assumes the existence of a *global stabilization time* (GST) and a bound δ such that after GST, every message sent by a correct process on a correct channel is received within δ time units of its transmission. Messages sent before GST may experience arbitrary delays. Additionally, the model assumes that processes have clocks that may drift unboundedly before GST, but do not drift thereafter. Both GST and δ are a priori unknown.

Let Value be an arbitrary domain of values. The consensus object provides a single operation $\text{propose}(x)$, $x \in \text{Value}$, which returns a

value in Value. The object has to satisfy the standard safety properties: all terminating propose() invocations must return the same value (Agreement); and propose() can only return a value passed to some propose() invocation (Validity). The following theorems are analogs of Theorems 1 and 2 for consensus.

THEOREM 5. *Let $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ be a generalized quorum system and $\tau : \mathcal{F} \rightarrow 2^{\mathcal{P}}$ be the termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) = U_f$. Then there exists an (\mathcal{F}, τ) -wait-free implementation of consensus.*

THEOREM 6. *Let \mathcal{F} be a fail-prone system and $\tau : \mathcal{F} \rightarrow 2^{\mathcal{P}}$ be the termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) \neq \emptyset$. If there exists an (\mathcal{F}, τ) -obstruction-free implementation of consensus, then there exist \mathcal{R} and \mathcal{W} such that $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ is a generalized quorum system. Moreover, for each $f \in \mathcal{F}$, we have $\tau(f) \subseteq U_f$.*

We first present a consensus protocol validating Theorem 5. To this end, we fix a generalized quorum system $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ and a termination mapping $\tau : \mathcal{F} \rightarrow 2^{\mathcal{P}}$ such that $\tau(f) = U_f$ holds for each $f \in \mathcal{F}$. As in §5, we assume without loss of generality that the connectivity relation of the graph $\mathcal{G} \setminus f$ is transitive for each $f \in \mathcal{F}$. Our protocol for consensus is shown in Figure 6.

Consensus vs registers under channel failures. Interestingly, solving consensus under process and channel failures is simpler than implementing registers. The main challenge we had to deal with when implementing registers was determining whether the information received by a process is up to date (§5). This is particularly difficult in the asynchronous model with unidirectional connectivity, where processes cannot rely on bidirectional exchanges to confirm the freshness of the information. In the partially synchronous model, however, processes can exploit the eventual timeliness of the network to determine freshness. Technically, this is done using a *view synchronizer* [1, 35], explained next. Then the connectivity stipulated by a generalized quorum system is sufficient to implement (\mathcal{F}, τ) -wait-free consensus using an algorithm similar to Paxos [30].

View synchronization. The consensus protocol works in a succession of views, each with a designated leader process $\text{leader}(v) = P_{((v-1) \bmod n)+1}$. Thus, the role of the leader rotates round-robin among the processes. The current view is tracked in a variable *view*. The protocol synchronizes views among processes via growing timeouts [1, 35]. Namely, each process spends the time $v \cdot C$ in view v , where C is an arbitrary positive constant. To ensure this, upon entering a view v , the process sets a timer *view_timer* for the duration $v \cdot C$ (line 29). When the timer expires, the process increments its view (line 28). Hence, the time spent by a process in each view grows monotonically as views increase. Even though processes do not communicate to synchronize their views, this simple mechanism ensures that all correct processes overlap for an arbitrarily long time in all but finitely many views.

PROPOSITION 2. *Let d be an arbitrary positive value. There exists a view \mathcal{V} such that for every view $v \geq \mathcal{V}$, all correct processes overlap in v for at least d .*

Protocol operation. A process stores its initial proposal in *my_val* (line 5), the last proposal it accepted in *val* and the view in which

```

1 view, aview ← 0, 0
2 val, my_val ← ⊥, ⊥
3 phase ∈ {ENTER, PROPOSE, ACCEPT, DECIDE}
4 function propose(x):
5   my_val ← x
6   async wait until phase = DECIDE
7   return val
8 when received {1B(view, vj, xj) | pj ∈ R} from some R ∈ R
9   pre: phase = ENTER
10  if ∀pj ∈ R. xj = ⊥ then
11    if my_val = ⊥ then return
12    send 2A(view, my_val) to all
13  else
14    let pj ∈ R be such that xj ≠ ⊥ ∧
      (∀pk ∈ R. xk ≠ ⊥ ⇒ vk ≤ vj)
15    send 2A(view, xj) to all
16  phase ← PROPOSE
17 when received 2A(view, x)
18  pre: phase ∈ {ENTER, PROPOSE}
19  val ← x
20  aview ← view
21  send 2B(view, x) to all
22  phase ← ACCEPT
23 when received {2B(view, x) | pj ∈ W} from some W ∈ W
24  val ← x
25  aview ← view
26  phase ← DECIDE
27 on startup or when the timer view_timer expires
28  view ← view + 1
29  start_timer(view_timer, view · C)
30  send 1B(view, aview, val) to leader(view)
31  phase ← ENTER

```

Figure 6: The consensus protocol at a process p_i .

this happened in *aview*. A variable *phase* tracks the progress of the process through the different phases of the protocol. When a process enters a view v , it sends the information about its last accepted value to $\text{leader}(v)$ in a 1B message (line 30). This message is analogous to the 1B message of Paxos; there is no analog of a 1A message, because leader election is controlled by the synchronizer.

A leader waits until it receives 1B messages from every member of some read quorum corresponding to its view (line 8); messages from lower views (considered out of date) are ignored. Based on these messages, the leader computes its proposal similarly to Paxos. If some process has previously accepted a value, the leader picks the one accepted in the maximal view. If there is no such value and propose() has already been invoked at the leader, the leader picks its own value. Otherwise, it skips its turn.

A process waits until it receives a 2A message from the leader of its view (line 17) and accepts the proposal by updating its *val* and *aview*. It then notifies every process about this through a 2B

message. Finally, when a process receives matching 2B messages from every member of some write quorum (line 23), it knows that a decision has been reached, and it sets `phase = DECIDE`. If there is an ongoing `propose()` invocation, this validates the condition at line 6, and the process returns the decision to the caller.

PROOF OF THEOREM 5. It is easy to see that the protocol satisfies Validity. The proof of Agreement is virtually identical to that of Paxos, relying on the Consistency property of the generalized quorum system. We now prove that the protocol is (\mathcal{F}, τ) -wait-free.

Fix a failure pattern $f \in \mathcal{F}$ and a process $p \in \tau(f)$ that invokes `propose(x)` for some x at a time t' . By Availability, there exist $W \in \mathcal{W}$ and $R \in \mathcal{R}$ such that W is f -available, and W is f -reachable from R . By Proposition 1, $W \subseteq U_f$. Then since $p \in \tau(f) = U_f$, p is strongly connected to all processes in W via channels correct under f . Thus, the following hold after GST: (i) R can reach p through timely channels; and (ii) p can exchange messages with every member of W via timely channels.

By Proposition 2 and since leaders rotate round-robin, there exists a view v led by p such that all correct processes enter v after $\max(\text{GST}, t')$ and overlap in this view for more than 3δ . We now show that this overlap is sufficient for p to reach a decision. Let t be the earliest time by which every correct process has entered v . Then no correct process leaves v until after $t + 3\delta$. When a process enters v , it sends a 1B message to p (line 30). By (i), p is guaranteed to receive 1B messages for view v from every member of R by the time $t + \delta$, thereby validating the guard at line 8. As a result, by this time p will send its proposal in a 2A message while still in v . By (ii), each process in W will receive this message no later than $t + 2\delta$, and respond with a 2B message that will reach p (line 17). Thus, by the time $t + 3\delta$, p is guaranteed to collect 2B messages for view v from every member of W (line 23). After this p sets `phase = DECIDE`, thus satisfying the guard at line 6 and deciding. \square

Lower bound. We now argue that Theorem 6 holds. First, note that the proof of the lower bound for lattice agreement (§6) remains valid in the partially synchronous model: since the execution σ constructed in the proof is finite, it is also valid under partial synchrony where all its actions occur before GST. Just like under asynchrony, the lower bound for lattice agreement directly implies the one for registers under partial synchrony [2, 11]. Finally, since consensus can be used to implement a MWMR atomic register, the lower bound for consensus follows from the bound for registers.

8 Related Work

Research on tolerating channel failures has primarily focused on consensus solvability in synchronous systems, where an adversary can disconnect channels in every round but cannot crash processes [3, 14, 15, 40, 43–45]. The seminal paper by Dolev [16] and subsequent work (see [42] for survey) explored this problem in general networks as a function of the network topology, process failure models, and synchrony constraints. However, this work considers only consensus formulations requiring termination at all correct processes, which in its turn requires them to be reliably connected. In contrast, we consider more general liveness conditions where termination is only required at specific subsets of correct processes. Our results demonstrate that this relaxation leads to a

much richer characterization of connectivity, which notably does not require all correct processes to be bidirectionally connected.

Early models, such as *send omission* [24] and *generalized omission* [41], extended crash failures by allowing processes to fail to send or receive messages. These models were shown to be computationally equivalent to crash failures in both synchronous [38] and asynchronous [13] systems. However, they are overly restrictive as they classify any non-crashed process with unreliable connectivity as faulty. In contrast, our approach allows correct processes to have unreliable connectivity. Santoro and Widmayer introduced the *mobile omission* failure model [43], which decouples message loss from process failures. They demonstrated that solving consensus in a synchronous round-based system without process failures requires the communication graph to contain a strongly connected component in every round [43, 44]. In contrast, our lower bounds show that implementing consensus – or even a register – in a partially synchronous system necessitates connectivity constraints that hold throughout the entire execution.

Failure detectors and consensus have been shown to be implementable in the presence of network partitions [4, 17, 18, 23] provided a majority of correct processes are strongly connected and can eventually communicate in a timely fashion. In contrast, we show that much weaker connectivity constraints, captured via generalized quorum systems, are necessary and sufficient for implementing both register and consensus. Aguilera et al. [6] and subsequent work [5, 20, 28, 32] studied the implementation of Ω – the weakest for consensus [12] – under various weak models of synchrony, link reliability, and connectivity. This work, however, mainly focused on identifying minimal timeliness requirements sufficient for implementing Ω while assuming at least fair-lossy connectivity between each pair of processes. Given that reliable channels can be implemented on top of fair-lossy ones, our results imply that these connectivity conditions are too strong. Furthermore, while the weak connectivity conditions of system S introduced in [6] were shown to be sufficient for implementing Ω , they were later proven insufficient for consensus [36].

In our previous work [36], we considered systems with process crashes and channel disconnections for $n = 2k + 1$, where any k processes can fail. We proved that a majority of reliably connected correct processes is necessary for implementing registers or consensus. This work, however, does not address solvability under arbitrary fail-prone systems, such as those with $k < \lfloor \frac{n-1}{2} \rfloor$ or those not based on failure thresholds [33, 34].

Alquraan et al. [8] presented a study of system failures due to faulty channels, which we already mentioned in §1. This work highlights the practical importance of designing provably correct systems that explicitly account for channel failures. Follow-up work [7, 39] proposed practical systems for tolerating channel failures, but did not investigate optimal fault-tolerance assumptions.

Acknowledgments

This work was partially supported by the projects BYZANTIUM, DECO and PRODIGY funded by MCIN/AEI, and REDONDA funded by the CHIST-ERA network. We thank Petr Kuznetsov for helpful comments and discussions.

References

- [1] Lăcrămioara Aștefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni, and Eugen Zălinescu. 2021. Tenderbake - A Solution to Dynamic Repeated Consensus for Blockchains. In *Symposium on Foundations and Applications of Blockchain (FAB)*.
- [2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. 1993. Atomic snapshots of shared memory. *J. ACM* 40, 4 (1993), 873–890.
- [3] Yehuda Afek and Eli Gafni. 2013. Asynchrony from Synchrony. In *International Conference on Distributed Computing and Networking (ICDCN)*.
- [4] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. 1999. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theor. Comput. Sci.* 220, 1 (1999), 3–30.
- [5] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. 2004. Communication-Efficient Leader Election and Consensus with Limited Link Synchrony. In *Symposium on Principles of Distributed Computing (PODC)*.
- [6] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. 2008. On implementing Omega in systems with weak reliability and synchrony assumptions. *Distributed Comput.* 21, 4 (2008), 285–314.
- [7] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *Symposium on Operating Systems Design and Implementation (OSDI)*.
- [8] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. 2018. An Analysis of Network-Partitioning Failures in Cloud Systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*.
- [9] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. 1995. Sharing Memory Robustly in Message-Passing Systems. *J. ACM* 42, 1 (1995), 124–142.
- [10] Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2016. Lower Bound on the Step Complexity of Anonymous Binary Consensus. In *Symposium on Distributed Computing (DISC)*.
- [11] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. 1995. Atomic snapshots using lattice agreement. *Distrib. Comput.* 8, 3 (1995), 121–132.
- [12] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. 1996. The weakest failure detector for solving consensus. *J. ACM* 43, 4 (1996), 685–722.
- [13] Brian Coan. 1988. A Compiler That Increases the Fault Tolerance of Asynchronous Protocols. *IEEE Trans. Comput.* 37, 12 (1988), 1541–1553.
- [14] Étienne Coulouma and Emmanuel Godard. 2013. A Characterization of Dynamic Networks Where Consensus Is Solvable. In *Structural Information and Communication Complexity (SIROCCO)*.
- [15] Étienne Coulouma, Emmanuel Godard, and Joseph Peters. 2015. A characterization of oblivious message adversaries for which Consensus is solvable. *Theoretical Computer Science* 584 (2015), 80–90.
- [16] Danny Dolev. 1982. The Byzantine Generals Strike Again. *J. Algorithms* 3, 1 (1982), 14–30.
- [17] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. 1996. *Failure detectors in omission failure environments*. Technical Report TR96-1608. Department of Computer Science, Cornell University.
- [18] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. 1997. Failure detectors in omission failure environments (Brief Announcement). In *Symposium on Principles of Distributed Computing (PODC)*.
- [19] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (1988), 288–323.
- [20] Antonio Fernández Anta and Michel Raynal. 2007. From an Intermittent Rotating Star to a Leader. In *Conference on Principles of Distributed Systems (OPODIS)*.
- [21] Faith Fich, Maurice Herlihy, and Nir Shavit. 1998. On the Space Complexity of Randomized Synchronization. *J. ACM* 45, 5 (1998), 843–862.
- [22] Colin Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. In *Australian Computer Science Conference (ASCS)*.
- [23] Roy Friedman, Idit Keidar, Dahlia Malkhi, Ken Birman, and Danny Dolev. 1995. *Deciding in partitionable networks*. Technical Report TR95-1554. Department of Computer Science, Cornell University.
- [24] Vassos Hadzilacos. 1983. *Byzantine agreement under restricted type of failures (not telling the truth is different from telling lies)*. Technical Report TR-18-63. Department of Computer Science, Harvard University.
- [25] Maurice Herlihy, Victor Luchangco, and Mark Moir. 2003. Obstruction-Free Synchronization: Double-Ended Queues as an Example. In *International Conference on Distributed Computing Systems (ICDCS)*.
- [26] Maurice Herlihy and Jeannette Wing. 1990. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.
- [27] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. 2016. Flexible Paxos: Quorum intersection revisited. In *Conference on Principles of Distributed Systems (OPODIS)*.
- [28] Martin Huttle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou. 2009. Chasing the Weakest System Model for Implementing Ω and Consensus. *IEEE Trans. Dependable Secur. Comput.* 6, 4 (2009), 269–281.
- [29] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.
- [30] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (1998), 133–169.
- [31] Nancy A. Lynch and Alexander A. Shvartsman. 2002. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In *Symposium on Distributed Computing (DISC)*.
- [32] Dahlia Malkhi, Florin Oprea, and Lidong Zhou. 2005. Ω Meets Paxos: Leader Election and Stability Without Eventual Timely Links. In *Symposium on Distributed Computing (DISC)*.
- [33] Dahlia Malkhi and Michael K. Reiter. 1998. Byzantine Quorum Systems. *Distributed Comput.* 11, 4 (1998), 203–213.
- [34] Moni Naor and Avishai Wool. 1994. The load, capacity and availability of quorum systems. In *Symposium on Foundations of Computer Science (FOCS)*.
- [35] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. 2020. Cogsworth: Byzantine View Synchronization. In *Cryptoeconomics Systems Conference (CES)*.
- [36] Alejandro Naser-Pastoriza, Gregory Chockler, and Alexey Gotsman. 2023. Fault-tolerant computing with unreliable channels. In *Conference on Principles of Distributed Systems (OPODIS)*.
- [37] Alejandro Naser-Pastoriza, Gregory Chockler, Alexey Gotsman, and Fedor Ryabinin. 2025. Tight Bounds on Channel Reliability via Generalized Quorum Systems (Extended Version). arXiv:2505.02646 [cs.DC] <https://arxiv.org/abs/2505.02646>
- [38] Gil Neiger and Sam Toueg. 1990. Automatically Increasing the Fault-Tolerance of Distributed Algorithms. *J. Algorithms* 11, 3 (1990), 374–419.
- [39] Harald Ng, Seif Haridi, and Paris Carbone. 2023. Omni-Paxos: Breaking the Barriers of Partial Connectivity. In *European Conference on Computer Systems (EuroSys)*.
- [40] Thomas Nowak, Ulrich Schmid, and Kyrill Winkler. 2019. Topological Characterization of Consensus under General Message Adversaries. In *Symposium on Principles of Distributed Computing (PODC)*.
- [41] Kenneth J. Perry and Sam Toueg. 1986. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Software Eng.* 12, 3 (1986), 477–482.
- [42] Dimitris Sakavalas and Lewis Tseng. 2019. *Network Topology and Fault-Tolerant Consensus*. Morgan & Claypool Publishers.
- [43] Nicola Santoro and Peter Widmayer. 1989. Time is not a healer. In *Symposium on Theoretical Aspects of Computer Science (STACS)*.
- [44] Nicola Santoro and Peter Widmayer. 1990. Distributed function evaluation in the presence of transmission faults. In *Symposium on Algorithms (SIGAL)*.
- [45] Ulrich Schmid, Bettina Weiss, and Idit Keidar. 2009. Impossibility Results and Lower Bounds for Consensus under Link Failures. *SIAM J. Comput.* 38, 5 (2009),