Fault-Tolerant Computing with Unreliable Channels

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

Gregory Chockler
University of Surrey, Guildford, UK

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Abstract

We study implementations of basic fault-tolerant primitives, such as consensus and registers, in message-passing systems subject to process crashes and a broad range of communication failures. Our results characterize the necessary and sufficient conditions for implementing these primitives as a function of the connectivity constraints and synchrony assumptions. Our main contribution is a new algorithm for partially synchronous consensus that is resilient to process crashes and channel failures and is optimal in its connectivity requirements. In contrast to prior work, our algorithm assumes the most general model of message loss where faulty channels are *flaky*, i.e., can lose messages without any guarantee of fairness. This failure model is particularly challenging for consensus algorithms, as it rules out standard solutions based on leader oracles and failure detectors. To circumvent this limitation, we construct our solution using a new variant of the recently proposed *view synchronizer* abstraction, which we adapt to the crash-prone setting with flaky channels.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Consensus, network partitions, liveness, synchronizers

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2023.21

Related Version *Extended Version*: `https://arxiv.org/abs/2305.15150`

Funding This work was partially supported by the PRODIGY and DECO projects funded by MCIN/AEI, the BLOQUES project funded by the Madrid regional government, and by a research grant from Nomadic Labs.

1 Introduction

We are concerned with implementing basic fault-tolerant primitives, such as registers and consensus, in systems where processes can crash and communication channels may lose messages. This setting is of practical importance, since channel failures regularly occur in real-world deployments [10, 15, 33]. They arise from a variety of reasons – physical infrastructure failures, bugs in switch software, configuration errors – and they often lead to system outages. For example, Alquraan et al. [7] conducted a comprehensive study of failures due to network partitions in widely used replicated data stores. It found that a majority of such failures led to catastrophic effects and that the resolution of almost half of these failures required redesigning a system mechanism. Thus, the failures were not simply due to coding bugs, but to design flaws.

Perhaps the most challenging aspect of real-world network failures is that the ways the network can break down can be arbitrarily complex, resulting in a wide variety of connectivity configurations. While in the simplest case individual channel failures may not affect the overall network connectivity (Figure 1a), in more complex scenarios the network may become partitioned into multiple components, some of which may end up connected in

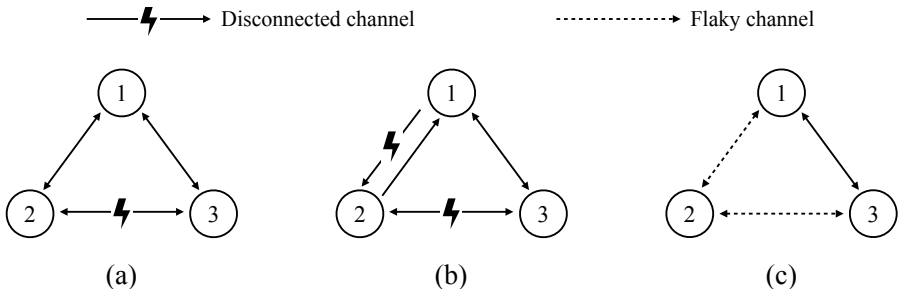

Figure 1 Examples of irregular connectivity configurations: (a) indirect connectivity; (b) asymmetric connectivity; and (c) flaky connectivity. All processes are correct.

one direction, but not in the other (Figure 1b). In the worst case, some components may become intermittently connected, causing an arbitrary subset of the messages transmitted between them to get lost (Figure 1c).

Intermittent connectivity has so far received little attention in the theory research. Most network models for studying consensus assume that channels are either (eventually) reliable or (eventually) disconnected (Figures 1a-b). Reliable channels are sometimes replaced by *fair lossy* ones, which only guarantee to deliver a message if it was sent infinitely many times. However, reliable and fair lossy channels are computationally equivalent: the former can be implemented from the latter by repeatedly resending each message until it gets acknowledged and filtering out duplicates [1].

Although the classical abstractions of failure or leader detectors [17] can be adapted to solve consensus in the above settings [4, 21, 22, 26, 35, 46], they are no longer useful in the presence of unconstrained message loss. Intuitively, the reason is that in this case they may fail to correctly identify processes with reliable connectivity, which is necessary to ensure the liveness of consensus. For example, suppose that the pattern of message loss experienced by the channels $2 \leftrightarrow 1$ and $2 \leftrightarrow 3$ in Figure 1c is such that all leader election messages are getting through. Thus, it is possible for process 2 to be elected as a leader. However, since any message sent by process 2 after being elected can be dropped (no matter how many times it is resent), the process may not be able to drive consensus to completion, violating liveness.

In this paper we investigate the possibility of implementing basic fault-tolerant abstractions resilient to a broad range of communication failures, including those induced by intermittent connectivity. To this end, we obtain several lower and upper bounds that characterize the solvability of registers and consensus as a function of the channel failure model, connectivity constraints and synchrony assumptions. Our main contribution is a new crash fault-tolerant algorithm that solves partially synchronous consensus under the *most general* model of message loss and is *optimal* in terms of its network connectivity requirements. Our algorithm furthermore offers a new way of constructing modular protocols for unreliable networks, which does not rely on the leader election or failure detection abstractions. Below we review our results in more detail.

Lower bounds. We first present a general framework for specifying fault-tolerance assumptions on process and channel failures (§2-3) by generalizing the classical notion of a *fail-prone system* [39]. For example, this framework allows us to specify whether an algorithm is meant to tolerate the failure patterns depicted in Figure 1. We then use this framework to establish minimal connectivity requirements necessary to implement registers or consensus (§4). To

Table 1 System models with different channel types and synchrony assumptions.

	Reliable/Disconnected (lower bound)	Eventually reliable/Flaky (upper bound)
Asynchronous (register)	$\mathcal{M}_{\text{ARD}}$	$\mathcal{M}_{\text{AEF}}$
Partially synchronous (consensus)	$\mathcal{M}_{\text{PRD}}$	$\mathcal{M}_{\text{PEF}}$

obtain strong lower bounds, we consider strong network models where correct channels are reliable and faulty channels are disconnected, i.e., drop all messages. We consider asynchrony for registers (model $\mathcal{M}_{\text{ARD}}$ in Table 1) and partial synchrony [23] for consensus (model $\mathcal{M}_{\text{PRD}}$): the execution starts with an asynchronous period and then becomes synchronous. The most interesting aspect of our lower bounds is that, unlike the prior work on consensus under unreliable connectivity (e.g., [2, 49]), they are proven for a very weak termination guarantee that only requires obstruction-free termination at a *subset* of processes. Informally, we show that for any n-process implementation of a register or consensus:

1. All processes where obstruction-freedom holds must be strongly connected via correct channels.
2. If the implementation tolerates k process crashes and $n = 2k + 1$, then any process where obstruction-freedom holds must belong to a set of $\geq k + 1$ correct processes strongly connected by correct channels.

We call the largest set satisfying the condition in (2) the *connected core* of the network. For example, in Figure 1a the connected core is $\{1, 2, 3\}$, whereas in Figures 1b-c it is $\{1, 3\}$.

The above results generalize the celebrated CAP theorem [14, 27], which says that it is impossible to guarantee all of Consistency, Availability and network Partition-tolerance (§4.1). Whereas CAP only says that to ensure availability and consistency, *some* channels must be correct, we establish *how many* are needed; and whereas CAP requires availability at *all* processes, we establish conditions required to ensure it only in *a part* of the system. Result (2) furthermore establishes stronger requirements for algorithms resilient to a minority of crashes (which is optimal [23, 37]). It shows that, even if obstruction-freedom is required only at a *single* process, a majority thereof must still be strongly connected by correct channels.

Upper bounds. The second part of our contribution is to propose algorithms for registers and consensus that match our lower bounds. These algorithms assume the existence of the connected core and guarantee wait-freedom at all of its members. They are designed to work in an adversarial model where correct channels are only *eventually* reliable and faulty channels are *flaky*, i.e., can drop any messages sent on them without any guarantee of fairness (models $\mathcal{M}_{\text{AEF}}$ and $\mathcal{M}_{\text{PEF}}$ in Table 1). Flakiness is a very broad failure mode that captures a number of failure patterns occurring in practice, including both full and intermittent loss of connectivity. It also subsumes the previously considered variants of lossy channels, such as eventually disconnected [4, 18] and fair lossy [1, 11]. Although irregular connectivity patterns have been studied before [3, 4, 21, 22, 26], to the best of our knowledge we are the first to propose register and consensus implementations in the presence of flaky channels.

The implementation for consensus is the more challenging one (§5). Since, as we explained above, failure and leader detectors are not useful in the presence of flaky channels, we take a different approach: we generalize the abstraction of a *view synchronizer* [12, 13, 40, 41], recently proposed for Byzantine consensus, to benign failure settings with flaky channels. Roughly, a view synchronizer facilitates a commonly used design pattern in which the protocol execution is divided into *views* (aka *rounds*). Each view has a designated leader that coordinates

the process interactions within that view. The task of the synchronizer is to ensure that sufficiently many correct processes are eventually able to spend sufficient time in the same view with a correct leader, which would then be able to drive consensus to completion. Supporting this functionality under partial synchrony is nontrivial, as during asynchronous periods clocks can diverge and messages used to synchronize views could get lost or delayed. View synchronizers encapsulate the necessary logic to deal with these complexities, thereby enabling modular design of consensus protocols.

In contrast to failure or leader detectors, a synchronizer does not attempt to identify the set of misbehaving processes (which, as we argued above, is impossible with flaky channels), but instead delegates this task to the top-level protocol. This can monitor the current leader using timeouts and other protocol-specific logic and ask the synchronizer to switch to another view with a different leader if it detects a lack of progress. For example, if processes 1 and 3 in Figure 1c do not observe progress on behalf of process 2 due to message loss, they would eventually initiate a view change and try another leader.

We present a specification of a view synchronizer sufficient to implement consensus in the presence of flaky channels, its implementation, and prove that the implementation satisfies our specification. Even though handling crashes is easier than Byzantine faults, flaky channels create another challenge: processes outside the connected core (such as process 2 in Figures 1b-c) falsely suspecting the current leader should not be able to force a view change, disrupting a working view. Using our synchronizer we then design a consensus protocol that tolerates flaky channels and prove its correctness.

Finally, we also demonstrate how our lower and upper bounds can be applied to establish consensus solvability in various existing models of weak connectivity [2, 3, 24, 32, 38]. In particular, we show that some connectivity models strong enough to implement the Ω leader detector are nevertheless too weak to implement consensus (§6).

2 System Model

We consider a set $\mathcal{P}$ of n processes which can 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 $(p, q) \in \mathcal{C}$ for sending messages from p to q.

We consider two notions of channel correctness (*reliable* and *eventually reliable*) and two notions of channel faultiness (*disconnected* and *flaky*). Given correct processes p and q:

- a channel (p, q) is *reliable* if it delivers every message sent by p to q;
- a channel (p, q) is *eventually reliable* if there exists a time t such that the channel delivers every message sent by p to q after t;
- a channel (p, q) is *disconnected* if it drops all messages sent by p to q; and
- a channel (p, q) is *flaky* if it drops an arbitrary subset of messages sent by p to q.

Flakiness is a very broad failure mode: it subsumes disconnections and allows the channel to choose which messages to drop, without any guarantee of fairness. Thus, flaky channels are strictly more permissive than fair lossy, which are computationally equivalent to reliable, as we explained in §1.

Our lower bounds assume the stronger notion of channel correctness (reliable) and the more restrictive notion of faultiness (disconnected), whereas our upper bounds assume weaker correctness (eventually reliable) and broader faultiness (flaky). We combine these channel reliability assumptions with two types of synchrony guarantees for correct channels, which yields four models used in our results (Table 1). We use the *asynchronous* model in our

results about registers, and the *partially synchronous* model [17, 23] in our results about consensus. The latter assumes that there exists a *global stabilization time* GST and a duration δ such that after GST message delays between correct processes connected by correct channels are bounded by δ. However, messages sent before GST can get arbitrarily delayed. Partial synchrony also assumes that processes have clocks that can drift unboundedly from the real time before GST, but do not drift thereafter. Both GST and δ are unknown to our protocols.

To state our results we need an ability to restrict which processes and channels can fail. We do this by generalizing the classical notion of a *fail-prone system* [39], which we formulate in a generic fashion irrespective of a specific channel failure type (flaky or disconnected). We specify the failure type when stating our lower or upper bounds. A *failure pattern* is a pair $(P, C) \in 2^{\mathcal{P}} \times 2^{\mathcal{C}}$ that defines which processes and channels fail in a single execution. We assume that C only contains channels between pairs of correct processes, since those incident to faulty processes fail automatically: $(p, q) \in C \implies \{p, q\} \cap P = \emptyset$. For a failure pattern $f = (P, C)$, an execution σ of the system is *f-compliant* if exactly the processes in P and the channels in C fail in σ. A *fail-prone system* $\mathcal{F}$ is a set of failure patterns. For example, a standard fail-prone system where any minority of processes can fail and channels between correct processes cannot fail corresponds to $\mathcal{F}_M = \{(Q, \emptyset) \mid Q \subseteq \mathcal{P} \wedge |Q| \leq \lfloor \frac{n-1}{2} \rfloor\}$.

3 Correctness Properties

We consider algorithms $\mathcal{A}$ that implement shared memory objects $\mathcal{O}$, such as a register or consensus, in the models just introduced. For an arbitrary domain of values *Val*, the *register* interface consists of operations *write*(v), $v \in$ *Val* and *read*(), which return *ack* and a value in *Val*, respectively. The interface of a consensus object consists of a single operation *propose*(v), $v \in$ *Val*, which returns a value in *Val*.

Safety. The consensus object is required to satisfy the standard safety properties: (*Agreement*) all terminating *propose* invocations must return the same value; and (*Validity*) *propose* can only return a value passed to some *propose* invocation.

We now specify the safety properties for registers. An operation op' *follows* an operation op, denoted $op \rightarrow op'$, if op' is invoked after op returns; op' is *concurrent* with op if neither $op \rightarrow op'$ nor $op' \rightarrow op$. The $\mathcal{O}$'s *sequential specification* specifies the behavior of $\mathcal{O}$ in the executions in which no operations are concurrent with each other. The sequential specification of a register requires every read to return the value written by the latest preceding write operation. An execution σ of $\mathcal{A}$ is *linearizable* [30] if there exists a set of responses X and a sequence $\pi = op_1, op_2, \ldots$ of all complete operations in σ and some subset of incomplete operations paired with responses in X such that π complies with $\mathcal{O}$'s sequential specification and satisfies $op_i \rightarrow op_j \implies i < j$. An execution σ of a register satisfies *safeness* if the subsequence of σ consisting of all write operations and all read operations not concurrent with any writes is linearizable. An implementation $\mathcal{A}$ of a register is *safe* [34] if each one of its executions satisfies safeness; $\mathcal{A}$ is *atomic* if each one of its executions is linearizable.

Liveness. Due to faulty channels, some correct processes may not be able to terminate, e.g., if they are partitioned off from the rest of the system. Therefore, to state liveness we parameterize the classical notions of obstruction-freedom and wait-freedom by 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 $f = (P, C)$, we say that $\mathcal{A}$ is (f, T)-*wait-free* if $T \subseteq \mathcal{P} \setminus P$ and for every process $p \in T$, operation *op* and f-compliant fair execution σ of $\mathcal{A}$, if *op* is invoked by p in σ, then *op* eventually returns. An operation *op eventually executes solo* in an execution σ if either *(i) op* returns in σ, or *(ii)* there exists a suffix σ' of σ such that for all operations op', if op' is concurrent with *op* in σ, then the process that invoked op' is crashed in σ'. We say that $\mathcal{A}$ is (f, T)-*obstruction-free* if $T \subseteq \mathcal{P} \setminus P$ and 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. Note that our notion of obstruction-freedom mirrors its well-known shared memory counterparts, such as solo termination [25] and the formalization of obstruction-freedom [31] given in [9].

We next lift these notions to a fail-prone system $\mathcal{F}$ and a *termination mapping* $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ – a function mapping each failure pattern to the set of correct processes whose operations are required to terminate (thus, $\forall f = (P, C) \in \mathcal{F}.\ \tau(f) \subseteq \mathcal{P} \setminus P$). Namely, we say that $\mathcal{A}$ is $(\mathcal{F}, \tau)$-*obstruction-free* (respectively, *wait-free*) if for every $f \in \mathcal{F}$, $\mathcal{A}$ is $(f, \tau(f))$-obstruction-free (respectively, wait-free). For example, the standard guarantee of wait-freedom under a minority of processes failures corresponds to $(\mathcal{F}_M, \tau_M)$-wait-freedom, where $\mathcal{F}_M$ is defined in §2 and τ_M is such that $\forall f = (P, \emptyset) \in \mathcal{F}_M.\, \tau_M(f) = \mathcal{P} \setminus P$.

4 Inherent Connectivity Requirements for Registers and Consensus

We now investigate which connectivity requirements are necessary to implement first registers and then consensus in the models of Table 1. We start with a few simple results that serve as a stepping stone for our later lower bounds. These results are also of independent interest because they generalize the celebrated CAP theorem [14]: it is impossible to guarantee all of Consistency, Availability and network Partition-tolerance.

4.1 CAP Revisited

The CAP formalization by Gilbert and Lynch [27] considers an asynchronous system without process failures. It models consistency as implementing an atomic register, availability as providing wait-freedom at all processes, and partition-tolerance as tolerating channels that can lose any messages (in our terminology, flaky).

▶ **Theorem 1** (CAP). *It is impossible to implement a wait-free atomic register in a message-passing system where processes do not fail, but all channels are flaky.*

Despite its fame, the CAP theorem yields only a weak lower bound. First, it is not tight: the theorem only says that, to implement a wait-free atomic register, *some* channels must be correct, but obviously, a single correct channel would not be sufficient in general. Second, CAP requires availability at *all* processes and thus is not applicable if we are trying to ensure availability at least in *a part* of the system, as formalized by the notions of liveness in §3. Our first result lifts these limitations and also strengthens CAP in other ways: it considers a weaker object, obstruction-free safe register, and a stronger system model $\mathcal{M}_{\text{ARD}}$, where channels can only fail by disconnection. Informally, we show that, in this setting, all processes where obstruction-freedom holds must be transitively connected via correct channels: no such process can be partitioned off from the rest.

Formally, let $\mathcal{G} = (\mathcal{P}, \mathcal{C})$ be the directed graph constructed by taking processes as vertices and channels as edges. For a failure pattern $f = (P, C)$, let $\mathcal{G} \setminus f$ denote the subgraph of $\mathcal{G}$ obtained by removing all processes in P along with their incident channels, as well as all channels in C.

▶ **Theorem 2.** *Let f be a failure pattern and $T \subseteq \mathcal{P}$. If some algorithm $\mathcal{A}$ implements an (f,T)-obstruction-free safe register over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), then T is strongly connected in $\mathcal{G} \setminus f$.*

Before proving the theorem, we note some of its consequences. The theorem yields the following corollary in the special case when the failure pattern f disallows process failures and obstruction-freedom is required at all processes.

▶ **Corollary 3.** *Let f be a failure pattern that disallows process failures: $\exists C \subseteq \mathcal{C}. f = (\emptyset, C)$. If some algorithm $\mathcal{A}$ implements an $(f,\mathcal{P})$-obstruction-free safe register over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), then the graph $\mathcal{G} \setminus f$ is strongly connected.*

This implies CAP (Theorem 1): if an algorithm $\mathcal{A}$ implements an atomic wait-free register, then it also implements an obstruction-free safe register, and by the above corollary, all processes must be strongly connected by correct channels. Corollary 3 also yields a tight lower bound. To see this, consider its lifting to an arbitrary fail-prone system $\mathcal{F}$ and the termination mapping $\tau_{\mathcal{P}} = \lambda f. \mathcal{P}$:

▶ **Corollary 4.** *Let $\mathcal{F}$ be a fail-prone system that disallows process failures: $\forall f \in \mathcal{F}. \exists C \subseteq \mathcal{C}. f = (\emptyset, C)$. If some algorithm $\mathcal{A}$ implements an $(\mathcal{F}, \tau_{\mathcal{P}})$-obstruction-free safe register over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), then for all $f \in \mathcal{F}$ the graph $\mathcal{G} \setminus f$ is strongly connected.*

Then the following proposition implies a matching upper bound: it shows that a wait-free atomic register can be implemented in the more adversarial model $\mathcal{M}_{\text{AEF}}$ (asynchronous / eventually reliable / flaky). The proposition follows from a more general result we present later (Theorem 8).

▶ **Proposition 5.** *Let $\mathcal{F}$ be a fail-prone system that disallows process failures and such that for all $f \in \mathcal{F}$, the graph $\mathcal{G} \setminus f$ is strongly connected. Then there exists an algorithm $\mathcal{A}$ implementing an $(\mathcal{F}, \tau_{\mathcal{P}})$-wait-free atomic register over $\mathcal{M}_{\text{AEF}}$ (asynchronous / eventually reliable / flaky).*

Proof of Theorem 2. Assume by contradiction that $\mathcal{A}$ is an (f,T)-obstruction-free implementation of a safe register, but T is not strongly connected in $\mathcal{G} \setminus f$. Then for some $u, v \in T$, either there is no path from u to v or from v to u in $\mathcal{G} \setminus f$. Without loss of generality we assume the former. Let S_u be the strongly connected component of $\mathcal{G} \setminus f$ containing u, and S_v the strongly connected component of $\mathcal{G} \setminus f$ containing v; then $S_u \cap S_v = \emptyset$. Let R_u be the set of processes outside S_u that can reach u in $\mathcal{G} \setminus f$, and R_v the set of processes outside S_v that can reach v (see Figure 2).

▷ Claim 1. For any $k \in \{u, v\}$, $R_k \cup S_k$ is unreachable from $\mathcal{P} \setminus (R_k \cup S_k)$ in $\mathcal{G} \setminus f$.

▷ Claim 2. For any $k \in \{u, v\}$, R_k is unreachable from S_k in $\mathcal{G} \setminus f$.

The above claims easily follow from the definitions of R_k and S_k.

▷ Claim 3. $S_u \cap (R_v \cup S_v) = \emptyset$.

Proof. Assume by contradiction that $S_u \cap (R_v \cup S_v) \neq \emptyset$ and let $w \in S_u \cap (R_v \cup S_v)$. Because $w \in S_u$, there exists a path from u to w. Because $w \in R_v \cup S_v$, there exists a path from w to v. By concatenating these two paths we get a path from u to v in $\mathcal{G} \setminus f$, contradicting our assumption that there is no such path. ◁

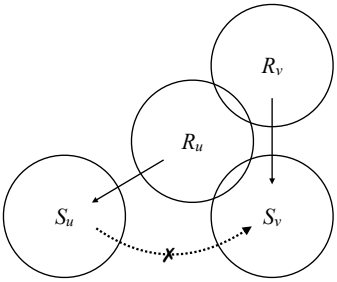

Figure 2 Illustration of the sets used in the proof of Theorem 2.

Let α_1 be a fair execution of $\mathcal{A}$ where processes and channels in f fail at the beginning, the process u invokes a *read* operation, and no other operation is invoked in α_1. Because $u \in T$ and $\mathcal{A}$ is (f,T)-obstruction-free, the *read* operation must eventually terminate. Since there are no *write* invocations, the *read* must return 0 – the initial value of the register. Let α_2 be the prefix of α_1 ending with this response. By Claim 1, $R_u \cup S_u$ is unreachable from $\mathcal{P} \setminus (R_u \cup S_u)$. Thus, the actions by processes in $R_u \cup S_u$ do not depend on those by processes in $\mathcal{P} \setminus (R_u \cup S_u)$. Then the projection of α_2 to actions by processes in $R_u \cup S_u$ is an execution of $\mathcal{A}$: $\alpha_3 = \alpha_2|_{R_u \cup S_u}$. By Claim 2, R_u is unreachable from S_u. Thus, the actions by processes in R_u do not depend on those by processes in S_u. Then $\alpha = \alpha_3|_{R_u}\alpha_3|_{S_u}$ is also an execution of $\mathcal{A}$. Notice that $\alpha_3|_{R_u}$ does not contain any operation invocation, but it contains steps taken by processes upon startup.

Let β_1 be a fair execution of $\mathcal{A}$ where processes and channels in f fail at the beginning. The execution β_1 starts with all the actions from $\alpha_3|_{R_u}$ followed by a *write*(1) invocation by process v, and no other operation is invoked in β_1. Because $v \in T$ and $\mathcal{A}$ is (f,T)-obstruction-free, the *write* operation must eventually terminate. Let β_2 be the prefix of β_1 ending with this response. By Claim 1, $R_u \cup S_u$ is unreachable from $\mathcal{P} \setminus (R_u \cup S_u)$. By Claim 2, R_u is unreachable from S_u. Therefore, R_u is unreachable from $\mathcal{P} \setminus R_u$. By Claim 1, $R_v \cup S_v$ is unreachable from $\mathcal{P} \setminus (R_v \cup S_v)$. Therefore, $R_u \cup R_v \cup S_v$ is unreachable from $\mathcal{P} \setminus (R_u \cup R_v \cup S_v)$. Thus, the actions by processes in $R_u \cup R_v \cup S_v$ do not depend on those by processes in $\mathcal{P} \setminus (R_u \cup R_v \cup S_v)$. Then $\beta = \beta_2|_{R_u \cup R_v \cup S_v}$ is an execution of $\mathcal{A}$. Recall that β_1 starts with $\alpha_3|_{R_u}$, and hence, so does β. Let δ be the suffix of β such that $\beta = \alpha_3|_{R_u}\delta$.

Consider the execution $\sigma = \alpha_3|_{R_u}\delta\alpha_3|_{S_u}$ where the actions occur before processes and channels in f fail. By Claim 3, $S_u \cap (R_v \cup S_v) = \emptyset$, and by the definition of R_u, we have $S_u \cap R_u = \emptyset$. Hence, $S_u \cap (R_u \cup R_v \cup S_v) = \emptyset$. Then, given that δ only contains actions by processes in $R_u \cup R_v \cup S_v$, we get: $\sigma|_{R_u \cup R_v \cup S_v} = (\alpha_3|_{R_u}\delta\alpha_3|_{S_u})|_{R_u \cup R_v \cup S_v} = \alpha_3|_{R_u}\delta = \beta$ and $\sigma|_{S_u} = (\alpha_3|_{R_u}\delta\alpha_3|_{S_u})|_{S_u} = \alpha_3|_{S_u} = (\alpha_3|_{R_u}\alpha_3|_{S_u})|_{S_u} = \alpha|_{S_u}$. Therefore, σ is indistinguishable from β to the processes in $R_u \cup R_v \cup S_v$ and from α to the processes in S_u. Finally, $\sigma|_{\mathcal{P} \setminus (R_u \cup S_u \cup R_v \cup S_v)} = \varepsilon$.

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 *write*(1) terminates before a *read* that fetches 0 is invoked. This contradicts the assumption that $\mathcal{A}$ implements a safe register. The contradiction shows that T must be strongly connected in $\mathcal{G} \setminus f$. ◀

4.2 Connectivity Requirements under Bounded Process Failures

Consider a straightforward lifting of Theorem 2 to fail-prone systems:

▶ **Corollary 6.** *Let $\mathcal{F}$ be a fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ a termination mapping. If some algorithm $\mathcal{A}$ implements an $(\mathcal{F}, \tau)$-obstruction-free safe register over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), then for all $f \in \mathcal{F}$, $\tau(f)$ is strongly connected in $\mathcal{G} \setminus f$.*

This result does not make any assumptions about the fail-prone system, and its CAP-like specialization given by Corollary 4 only considers fail-prone systems without process failures. However, algorithms are commonly designed to tolerate a bounded number of process failures, and we next show that stronger connectivity requirements are necessary in this case. To formalize this class of algorithms, we use the following notion. A *k-fail-prone system* $\mathcal{F}$ allows any set of k processes or fewer to fail, but disallows failures of more than k processes:

$$(\forall P \subseteq \mathcal{P}.\ |P| \leq k \implies \exists C \subseteq \mathcal{C}.\ (P, C) \in \mathcal{F}) \wedge (\forall (P, C) \in \mathcal{F}.\ |P| \leq k).$$

For example, the system $\mathcal{F}_M$ from §2 is $\lfloor \frac{n-1}{2} \rfloor$-fail-prone.

The following theorem establishes minimal connectivity constraints required to implement a safe register under the model $\mathcal{M}_{\text{ARD}}$ in the presence of k process crashes. It assumes a particularly weak termination guarantee which only requires obstruction-freedom to hold at some *non-empty* set of processes for each failure pattern. The theorem states that, no matter how small this set is, it must be part of a set of $> k$ correct processes strongly connected by correct channels.

▶ **Theorem 7.** *Let $\mathcal{F}$ be a k-fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ a termination mapping such that $\forall f = (P, C) \in \mathcal{F}. \tau(f) \neq \emptyset$. Assume that some algorithm $\mathcal{A}$ implements an $(\mathcal{F}, \tau)$-obstruction-free safe register over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected). Then for all $f \in \mathcal{F}$, there exists a strongly connected component of $\mathcal{G} \setminus f$ that contains $\tau(f)$ and has a cardinality greater than k.*

Proof. Let $f \in \mathcal{F}$ and S be the strongly connected component of $\mathcal{G} \setminus f$ containing $\tau(f)$. Assume by contradiction that $|S| \leq k$. Pick an arbitrary process $p \in \tau(f)$; then p is correct according to f. Let α_1 be a fair execution of $\mathcal{A}$ where processes and channels in f fail at the beginning, the process p invokes a *read* operation, and no other operation is invoked in α_1. Because $p \in \tau(f)$ and $\mathcal{A}$ is $(\mathcal{F}, \tau)$-obstruction-free, the *read* operation must eventually terminate. Since there are no *write* invocations, the *read* must return 0 – the initial value of the register. Let α_2 be the prefix of α_1 ending with this response. Let R be the set of processes outside S that can reach S in $\mathcal{G} \setminus f$. Similarly to the proof of Theorem 2, the definitions of R and S imply the following claims.

▷ Claim 1. $R \cup S$ is unreachable from $\mathcal{P} \setminus (R \cup S)$ in $\mathcal{G} \setminus f$.

▷ Claim 2. R is unreachable from S in $\mathcal{G} \setminus f$.

Claim 1 implies that the actions by processes in $R \cup S$ do not depend on those by processes in $\mathcal{P} \setminus (R \cup S)$. Then $\alpha_3 = \alpha_2|_{R \cup S}$ is an execution of $\mathcal{A}$. Claim 2 implies that the actions by processes in R do not depend on those by processes in S. Then $\alpha = \alpha_3|_R \alpha_3|_S$ is also an execution of $\mathcal{A}$.

Since $|S| \leq k$ and $\mathcal{F}$ is a k-fail-prone system, there exists $C' \subseteq \mathcal{C}$ such that $f' = (S, C') \in \mathcal{F}$. Pick an arbitrary process $q \in \tau(f')$; then $q \notin S$. Let β_1 be a fair execution of $\mathcal{A}$ where processes and channels in f' fail at the beginning. The execution β_1 starts with all the actions from $\alpha_3|_R$ followed by a *write*(1) invocation by process q, and no other operation is invoked in β_1. Because $q \in \tau(f')$ and $\mathcal{A}$ is $(\mathcal{F}, \tau)$-obstruction-free, the *write* operation must eventually terminate. Let β be the prefix of β_1 ending with this response and let δ be the suffix of β such that $\beta = \alpha_3|_R \delta$.

Consider the execution $\sigma = \alpha_3|_R \delta \alpha_3|_S$ where the actions occur before processes and channels in f fail. Given that δ does not contain any actions by processes in S, we get: $\sigma|_S = \alpha_3|_S = (\alpha_3|_R \alpha_3|_S)|_S = \alpha|_S$ and $\sigma|_{\mathcal{P}\setminus S} = \alpha_3|_R \delta = \beta$. Therefore, σ is indistinguishable from α to the processes in S and from β to the processes in $\mathcal{P} \setminus S$.

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 *write*(1) terminates before a *read* that fetches 0 is invoked. This contradicts the assumption that $\mathcal{A}$ implements a safe register. The contradiction derives from assuming that $|S| \leq k$, so that $|S| > k$. ◀

Theorem 7 is most interesting in the common practical case of $n = 2k + 1$, which is the minimal number of processes needed to tolerate k crashes in asynchronous registers [37] and partially synchronous consensus [23]. In this case the theorem ensures that for each failure pattern f, the graph $\mathcal{G} \setminus f$ has a strongly connected component containing $\geq k + 1$ processes. More generally, for arbitrary $\mathcal{G}$ and f, we call a strongly connected component of $\mathcal{G} \setminus f$ containing a majority of processes in $\mathcal{G}$ a *connected core* of the graph. It is easy to see there can exist at most one connected core for given $\mathcal{G}$ and f. For example, in Figure 1a the connected core is $\{1, 2, 3\}$, whereas in Figures 1b-c it is $\{1, 3\}$. As we now show, the lower bound of Theorem 7 is tight for $n = 2k + 1$. In fact, assuming the existence of a connected core, we can implement an atomic register that is wait-free at all members of the connected core under the more adversarial model $\mathcal{M}_{\text{AEF}}$ (asynchronous / eventually reliable / flaky).

▶ **Theorem 8.** *Let $\mathcal{F}$ be a fail-prone system such that for all $f \in \mathcal{F}$, the graph $\mathcal{G} \setminus f$ contains a connected core $\mathcal{S}_f$, and let $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be the termination mapping such that $\forall f \in \mathcal{F}$. $\tau(f) = \mathcal{S}_f$. Then there exists an $(\mathcal{F}, \tau)$-wait-free implementation of an atomic register over the model $\mathcal{M}_{\text{AEF}}$ (asynchronous / eventually reliable / flaky).*

We defer the proof of the theorem to [42, §A]. The proof constructs the desired implementation as a variant of ABD [8] that uses gossip-style data propagation to deal with indirect connectivity. The most interesting aspect of this implementation is that, despite the need for gossip, it uses only bounded space. We illustrate the technique for bounding space when presenting our consensus implementation in §5.

4.3 Connectivity Requirements for Consensus

The lower bounds in the previous section also apply to consensus under partial synchrony, with analogs of Theorems 2 and 7 formulated as follows:

▶ **Theorem 9.** *Let f be a failure pattern and $T \subseteq \mathcal{P}$. If some algorithm $\mathcal{A}$ is an (f, T)-obstruction-free implementation of consensus over $\mathcal{M}_{\text{PRD}}$ (partially synchronous / reliable / disconnected), then T is strongly connected in $\mathcal{G} \setminus f$.*

▶ **Theorem 10.** *Let $\mathcal{F}$ be a k-fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be a termination mapping such that $\forall f = (P, C) \in \mathcal{F}$. $\tau(f) \neq \emptyset$. Assume that some algorithm $\mathcal{A}$ is an $(\mathcal{F}, \tau)$-obstruction-free implementation of consensus over $\mathcal{M}_{\text{PRD}}$ (partially synchronous / reliable / disconnected). Then for all $f \in \mathcal{F}$, there exists a strongly connected component of $\mathcal{G} \setminus f$ that contains $\tau(f)$ and has a cardinality greater than k.*

To see why these theorems hold observe first that Theorems 2 and 7 also hold if the model $\mathcal{M}_{\text{ARD}}$ is replaced with $\mathcal{M}_{\text{PRD}}$: since the executions σ constructed in the proofs of the theorems are finite, they are also valid executions under $\mathcal{M}_{\text{PRD}}$ where all actions occur before GST. Then the required follows from the fact that registers can be implemented from consensus. We formally prove this for our setting in [42, §B].

Finally, similarly to Theorem 8, for the case of $n = 2k + 1$ we can prove an upper bound matching Theorem 10 in the model $\mathcal{M}_{\text{PEF}}$ (partially synchronous / eventually reliable / flaky). This result is much more difficult than the upper bound for registers, and we prove it in the next section.

▶ **Theorem 11.** *Let $\mathcal{F}$ be a fail-prone system such that for all $f \in \mathcal{F}$, the graph $\mathcal{G} \setminus f$ contains a connected core $\mathcal{S}_f$, and let $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be the termination mapping such that $\forall f \in \mathcal{F}.\ \tau(f) = \mathcal{S}_f$. Then there exists an $(\mathcal{F}, \tau)$-wait-free implementation of consensus over the model $\mathcal{M}_{\text{PEF}}$ (partially synchronous / eventually reliable / flaky).*

5 Consensus in the Presence of Flaky Channels

We now present a consensus protocol in the model $\mathcal{M}_{\text{PEF}}$ that validates Theorem 11. Consider a fail-prone system $\mathcal{F}$ satisfying the conditions of the theorem: for each $f \in \mathcal{F}$, the graph $\mathcal{G} \setminus f$ contains a connected core. For the remainder of the section we fix a failure pattern $f \in \mathcal{F}$ and let $\mathcal{S}$ be the corresponding connected core. Since, as we explained in §1, classical failure and leader detectors are not useful in the presence of flaky channels, we take a different approach. Our protocol is implemented on top of a *view synchronizer* [12, 13, 40, 41], which enables the processes to divide their execution into a series of views, each with a designated leader. At a high level, the synchronizer's goal is to bring sufficiently many correct processes with enough connectivity (e.g., those from $\mathcal{S}$) into a view led by a well-connected leader and keep them in that view for sufficiently long to reach an agreement. Supporting this in the presence of flaky channels is nontrivial as the processes outside the connected core (such as process 2 in Figures 1b-c) may fail to observe progress on behalf of the leader of a functional view and request a premature view change. We first present the specification (§5.1) and the implementation (§5.2) of a synchronizer that addresses this challenge. We then use it to construct a consensus protocol (§5.3) satisfying the requirements of Theorem 11.

5.1 Synchronizer Specification

We consider a synchronizer interface defined in [12, 40]. Let $\mathsf{View} = \{1, 2, \dots\}$ be the set of *views*, ranged over by v; we use 0 to denote an invalid initial view. The synchronizer produces notifications $\mathsf{new\_view}(v)$ at a process, telling it to enter a view v. To trigger these, the synchronizer allows a process to call a function $\mathsf{advance}()$, which signals that the process wishes to *advance* to a higher view. We assume that a process does not call $\mathsf{advance}()$ twice without an intervening $\mathsf{new\_view}$ notification.

In Figure 3 we give a specification of a view synchronizer for the system model $\mathcal{M}_{\text{PEF}}$, which is an adaptation of the one for the Byzantine setting [12]. The Monotonicity property ensures that, at any given process, its view can only increase. The Validity property ensures that a process may only enter a view $v + 1$ if some process in the connected core $\mathcal{S}$ has called $\mathsf{advance}$ in v. This prevents processes with bad connectivity from disrupting $\mathcal{S}$ by forcing view changes (e.g., process 2 in Figure 1c, where $\mathcal{S} = \{1, 3\}$). The Bounded Entry property ensures that, if some process from $\mathcal{S}$ enters a view v, then all processes from $\mathcal{S}$ will do so at most d units of time of each other (for some constant d). This only holds if within d no process from $\mathcal{S}$ attempts to advance to a higher view, as this may make some processes from $\mathcal{S}$ skip v and enter a higher view directly. Bounded Entry only holds starting from some view $\mathcal{V}$, since a synchronizer may not be able to guarantee it for views entered before GST. The Startup property ensures that if more than $\frac{n}{2}$ processes from $\mathcal{S}$ attempt to advance from view 0, then some process from $\mathcal{S}$ will enter view 1. The Progress property determines the

- **Monotonicity.** A process may only enter increasing views:
  $\forall i, v, v'.\ E_i(v)\downarrow\ \wedge\ E_i(v')\downarrow\ \implies\ (v < v'\ \iff\ E_i(v) < E_i(v'))$
- **Validity.** A process only enters $v+1$ if some process from $\mathcal{S}$ has attempted to advance from v:
  $\forall i, v.\ E_i(v+1)\downarrow\ \implies\ A^{\mathcal{S}}_{\text{first}}(v)\downarrow\ \wedge\ A^{\mathcal{S}}_{\text{first}}(v) < E_i(v+1)$
- **Bounded Entry.** For some $\mathcal{V}$ and d, if a process from $\mathcal{S}$ enters $v \geq \mathcal{V}$ and no process from $\mathcal{S}$ attempts to advance to a higher view within d, then every process from $\mathcal{S}$ will enter v within d:
  $\exists \mathcal{V}, d.\ \forall v \geq \mathcal{V}.\ E^{\mathcal{S}}_{\text{first}}(v)\downarrow\ \wedge\ \neg(A^{\mathcal{S}}_{\text{first}}(v) < E^{\mathcal{S}}_{\text{first}}(v) + d)\ \implies$
  $(\forall p_i \in \mathcal{S}.\ E_i(v)\downarrow)\ \wedge\ (E^{\mathcal{S}}_{\text{last}}(v) \leq E^{\mathcal{S}}_{\text{first}}(v) + d)$
- **Startup.** If $> \frac{n}{2}$ processes from $\mathcal{S}$ invoke advance, then some process from $\mathcal{S}$ will enter view 1:
  $(\exists P \subseteq \mathcal{S}.\ |P| > \frac{n}{2}\ \wedge\ (\forall p_i \in P.\ A_i(0)\downarrow))\ \implies\ E^{\mathcal{S}}_{\text{first}}(1)\downarrow$
- **Progress.** If a process from $\mathcal{S}$ enters v and, for some set $P \subseteq \mathcal{S}$ of $> \frac{n}{2}$ processes, any process in P that enters v eventually invokes advance, then some process from $\mathcal{S}$ will enter $v+1$:
  $\forall v.\ E^{\mathcal{S}}_{\text{first}}(v)\downarrow\ \wedge\ (\exists P \subseteq \mathcal{S}.\ |P| > \frac{n}{2}\ \wedge\ (\forall p_i \in P.\ E_i(v)\downarrow\ \implies\ A_i(v)\downarrow))\ \implies\ E^{\mathcal{S}}_{\text{first}}(v+1)\downarrow$

Notation:

- $E_i(v)$: the time when process p_i enters a view v
- $E^{\mathcal{S}}_{\text{first}}(v)$, $E^{\mathcal{S}}_{\text{last}}(v)$: the earliest and the latest time when a process from $\mathcal{S}$ enters a view v
- $A_i(v)$, $A^{\mathcal{S}}_{\text{first}}(v)$, $A^{\mathcal{S}}_{\text{last}}(v)$: similarly for times of attempts to advance from a view v
- $g(x)\downarrow$, $g(x)\uparrow$: $g(x)$ is defined/undefined

Figure 3 Synchronizer properties satisfied in executions with connected core $\mathcal{S}$.

conditions under which some process from $\mathcal{S}$ will enter a view $v+1$: this will happen if a process from $\mathcal{S}$ enters the view v, and for some set P of more than $\frac{n}{2}$ processes from $\mathcal{S}$, any process in P entering v eventually invokes advance (e.g., 1 and 3 in Figure 1c).

As we show below, the above properties work in tandem to ensure the liveness of consensus in the presence of flaky channels. Informally, Progress allows processes to iterate over views in search for one with a well-connected leader; Bounded Entry enables all processes in the connected core to promptly enter this view; and Validity ensures that these processes can stay in it despite any disruption from processes with flaky connectivity.

5.2 Synchronizer Implementation

In Figure 4 we present an algorithm that implements the specification in Figure 3 in the model $\mathcal{M}_{\text{PEF}}$. This implementation requires only bounded space, despite the fact that correct channels in $\mathcal{M}_{\text{PEF}}$ are only *eventually* reliable, and thus can lose messages before GST. A process stores its current view in a variable curr_view. A process also maintains an array views tracking, for every other process, the highest view to which it wishes to advance. When the process invokes advance (line 1), the synchronizer does not immediately switch to the next view. Instead, it updates its entry in the views array and propagates the whole array in a `WISH` message, advertising its wish to advance (line 3). Upon the receipt of a `WISH` message (line 6), a process incorporates the information received into its views array, keeping entries with the highest view (line 7). This mechanism ensures that information is propagated between processes that do not have direct connectivity via a correct channel.

Since the membership of $\mathcal{S}$ is unknown to the processes, to satisfy Validity the synchronizer cannot initiate a view change based on an advance() call by a single process: the synchronizer cannot tell whether this process is from $\mathcal{S}$ or not. Instead, we require wishes to advance from

```
1  function advance():
2      views[i] ← curr_view + 1
3      send WISH(views) to all

4  periodically                ▷ every ρ time units
5      send WISH(views) to all

6  when received WISH(V)
7      for p_j ∈ P do views[j] ← max(views[j], V[j])
8      v' ← max{v | ∃p_j. views[j] = v ∧ |{p_k | views[k] ≥ v}| > n/2}
9      if v' > curr_view then
10         curr_view ← v'
11         trigger new_view(v')
12         send WISH(views) to all
```

Figure 4 Synchronizer at a process p_i.

a majority of processes, so that at least one of them must be from $\mathcal{S}$. In more detail, upon receiving a WISH message, we compute v' as the $(\lfloor \frac{n}{2} \rfloor + 1)$-st highest view in views (line 8). Thus, at least one process from $\mathcal{S}$ wishes to advance to a view $\geq v'$. In this case the current process enters v' if this view is greater than its curr_view (line 11). Note that a process may be forced to switch views even if it did not invoke advance; this helps lagging processes to catch up. To satisfy Bounded Entry, the process disseminates the information that made it enter the new view (line 12) to ensure that other processes also do so promptly. Finally, to deal with message loss before GST, a process periodically resends its views array (line 4).

We can also prove that the synchronizer satisfies Progress: this property requires $> \frac{n}{2}$ advance calls by processes in $\mathcal{S}$, which are well-connected enough for the corresponding WISHes to eventually propagate within $\mathcal{S}$ and enable the guard at line 9 (e.g., processes 1 and 3 in Figure 1c). Note that Progress wouldn't hold if it required $> \frac{n}{2}$ advance calls by any processes, not necessarily in $\mathcal{S}$ (e.g., processes 1 and 2): in this case we wouldn't be able to guarantee that all the corresponding WISHes eventually propagate. We defer the proof of correctness of the synchronizer to [42, §C].

▶ **Theorem 12.** *Let $\mathcal{F}$ be a fail-prone system such that for each $f \in \mathcal{F}$, $\mathcal{G} \setminus f$ contains a connected core. Then for any $f \in \mathcal{F}$ with an associated connected core $\mathcal{S}$, every f-compliant fair execution of the algorithm in Figure 4 over the model $\mathcal{M}_{\text{PEF}}$ satisfies the properties in Figure 3.*

5.3 Consensus Protocol

In Figure 5 we present a consensus protocol in the model $\mathcal{M}_{\text{PEF}}$ (partially synchronous / eventually reliable / flaky) that validates Theorem 11. The protocol is a variation of single-decree Paxos [35] where liveness is ensured with the help of a view synchronizer. Thus, the protocol works in a succession of views produced by the synchronizer. Each view v has a fixed leader $\mathsf{leader}(v) = p_{((v-1) \mod n)+1}$ responsible for proposing a value to the other processes, which vote on the proposal. Processes monitor the leader's behavior and ask the synchronizer to advance to another view if they suspect that the leader is faulty or has a bad connectivity.

State and communication. A process stores its current view in a variable view. A variable phase tracks the progress of the process through different phases of the protocol. The initial proposal is stored in my_proposal (line 4). The process also maintains the last proposal it accepted from a leader in val, and the view in which this happened in cview.

```
1  on startup
2    advance()

3  function propose(x):
4    my_proposal ← x
5    wait until phase = DECIDED
6    return val

7  on new_view(v)
8    view ← v
9    start_timer(decision_timer, timeout)
10   M1B[i] ← (view, cview, val)
11   phase ← ENTERED

12 when the timer decision_timer expires
13   timeout ← timeout + γ
14   advance()

15 periodically                ▷ every ρ time units
16   send STATE(M1B, M2A, M2B) to all

17 when received STATE(V1B, V2A, V2B)
18   for p_j ∈ P do
19     if V1B[j].view > M1B[j].view then
20       M1B[j] ← V1B[j]
21     if V2A[j].view > M2A[j].view then
22       M2A[j] ← V2A[j]
23     if V2B[j].view > M2B[j].view then
24       M2B[j] ← V2B[j]

25 when phase = ENTERED ∧ leader(view) = p_i ∧
   |{p_j | p_j ∈ P ∧ M1B[j].view = view}| > n/2
26   Q ← {p_j | p_j ∈ P ∧ M1B[j].view = view}
27   if ∀p_j. p_j ∈ Q ⟹ M1B[j].val = ⊥ then
28     if my_proposal = ⊥ then return
29     M2A[i] ← (view, my_proposal)
30   else
31     let p_j ∈ Q be such that
         M1B[j].val ≠ ⊥ ∧
         ∀p_k ∈ Q. M1B[k].cview ≤ M1B[j].cview
32     M2A[i] ← (view, M1B[j].val)
33   phase ← PROPOSED

34 when phase ∈ {ENTERED, PROPOSED} ∧
   p_l = leader(view) ∧ M2A[l].view = view
35   (cview, val) ← M2A[l]
36   M2B[i] ← (cview, val)
37   phase ← ACCEPTED

38 when ∃v, x. v ≥ view ∧
   |{p_j | p_j ∈ P ∧ M2B[j] = (v, x)}| > n/2
39   val ← x
40   stop_timer(decision_timer)
41   phase ← DECIDED
```

Figure 5 Consensus protocol at a process p_i.

Processes exchange messages analogous to the 1B, 2A and 2B messages from Paxos, each tagged with the view where the message was issued. There is no analog of 1A messages, because leader election is controlled by the synchronizer. Since correct processes may not be directly connected by correct channels, each process has to forward the information it receives from others. Since correct channels in $\mathcal{M}_{\text{PEF}}$ are only *eventually* reliable, this furthermore has to be repeated periodically. If implemented naively, this would require unbounded space to store all the messages that need to be forwarded. Instead, we observe that it is sufficient to store, for each message type and sender, only the message of this type received from this sender with the highest view. These are stored in the arrays M1B, M2A and M2B.

For simplicity, the pseudocode in Figure 5 separates computation from communication. Most of the handlers do not send messages, but instead just modify the arrays M1B, M2A and M2B. Then instead of sending individual 1B, 2A or 2B messages like in Paxos, a process periodically sends whole arrays M1B, M2A and M2B in one big STATE message (line 16). Upon receipt of a STATE message (line 17), a process incorporates the information received into its arrays M1B, M2A and M2B, keeping entries with the highest view. This mechanism ensures that information is propagated between processes that do not have direct connectivity while using only bounded space.

Normal protocol operation. When the synchronizer tells a process to enter a new view v (line 7), the process sets view $= v$ and writes the information about the last value it accepted into its entry in the M1B array. This information will be propagated to the leader of the

view as described above. A leader waits until its M1B array contains a majority of entries corresponding to its view (line 25). Based on these, the leader computes its proposal and stores it into its entry of the M2A array. The computation is done similarly to Paxos. If some process has previously accepted a value, the leader picks the value accepted in the maximal view (line 31). Otherwise, the leader is free to propose its own value. If propose() has already been invoked at the leader, it selects my_proposal (line 29). If this has not happened yet, the leader skips its turn (line 28).

Each process waits until its M2A array contains a proposal by the leader of its view (line 34). The process then accepts the proposal by updating its val and cview. It also notifies all other processes about this by storing the information about the accepted value into its entry of the M2B array. Finally, once a process has a majority of matching entries in its M2B array (line 38), it knows that the decision has been reached, and it sets phase = DECIDED. If there is an ongoing propose() invocation, the condition at line 5 is then satisfied and the process returns the decision to the client.

Triggering view changes. We now describe when a process invokes advance(), which is key to ensuring liveness. This occurs either on startup (line 2) or when the process suspects that the current leader is faulty or has bad connectivity. To this end, when a process enters a view, it sets a decision_timer for a duration timeout (line 9) and stops the timer when a decision is reached (line 40). If the timer expires before this, the process invokes advance (line 14). A process may wrongly suspect a good leader if the timeout is initially set too low with respect to the message delay δ, unknown to the process. To deal with this, a process increases timeout whenever the timer expires (line 13).

Correctness. It remains to prove that the algorithm in Figure 5 validates Theorem 11. The proof of the safety properties of consensus is virtually identical to that of Paxos [35]. Hence, we focus on proving liveness. Here we present a proof sketch that highlights the use of the synchronizer specification and defer the proofs of auxiliary lemmas to [42, §D].

Fix a failure pattern f and let $\mathcal{S}$ be the corresponding connected core guaranteed to exist by the assumptions of Theorem 11. We prove liveness by showing that the protocol establishes properties reminiscent of those of failure detectors [17]. First, similarly to their *completeness* property, we prove that every correct process eventually attempts to advance from a view where no progress is possible (e.g., because the leader is faulty or has insufficient connectivity). We say that a process p_i *decides* in a view v if it executes line 41 while having view $= v$.

▶ **Lemma 13.** *If a correct process p_i enters a view v, never decides in v and never enters a view higher than v, then p_i eventually invokes* advance *in v.*

Informally, the lemma holds because each process monitors the progress of a view using decision_timer.

Our next lemma is similar to the *eventual accuracy* property of failure detectors. It shows that if the timeout values are high enough, then eventually any process in $\mathcal{S}$ that enters a view where progress is possible (the leader is correct and has sufficient connectivity) will not attempt to advance from it. Formally, let diameter($\mathcal{S}$) be the longest distance in the graph $\mathcal{G} \setminus f$ between two vertices in $\mathcal{S}$. Also, let $\mathcal{V}$ and d be the view and the time duration for which Bounded Entry holds (Figure 3), and let $\mathsf{timeout}_i(v)$ be the value of timeout at the process p_i while in view v.

▶ **Lemma 14.** *Let $v \geq \mathcal{V}$ be a view such that* leader$(v) \in \mathcal{S}$, $E^{\mathcal{S}}_{\text{first}}(v) \geq$ GST *and* leader(v) *invokes* propose *no later than* $E^{\mathcal{S}}_{\text{first}}(v)$. *If at each process $p_i \in \mathcal{S}$ that enters v we have* timeout$_i(v) > d + 3(\delta + \rho)$diameter$(\mathcal{S})$, *then no process in $\mathcal{S}$ invokes* advance *in v.*

Intuitively, the lemma holds because Bounded Entry ensures that all processes from $\mathcal{S}$ will enter v promptly; then since the timeouts are high enough, processes will have sufficient time to exchange the messages needed to reach a decision and stop the timers.

Proof sketch for Theorem 11. By contradiction, assume there exists $f \in \mathcal{F}$, $p_j \in \tau(f) = \mathcal{S}$ and an f-compliant fair execution of the algorithm in Figure 5 such that p_j invokes propose at a time t, but the operation never returns. Using Progress and Lemma 13, we first prove that in this case the protocol keeps moving through views forever:

▷ Claim 1. Every view is entered by some process in $\mathcal{S}$.

We next prove the following:

▷ Claim 2. Every process in $\mathcal{S}$ executes the timer expiration handler at line 12 infinitely often.

Since a process increases timeout every time decision_timer expires, by Claim 2 all processes will eventually have timeout $> d + 3(\delta + \rho)$diameter$(\mathcal{S})$. Since leaders rotate round-robin, by Claim 1 there will be infinitely many views led by p_j. Hence, there exists a view $v_1 \geq \mathcal{V}$ led by p_j such that $E^{\mathcal{S}}_{\text{first}}(v_1) \geq \max\{\mathsf{GST}, t\}$ and for any process $p_i \in \mathcal{S}$ that enters v_1 we have timeout$_i(v_1) > d + 3(\delta + \rho)$diameter$(\mathcal{S})$. Then by Lemma 14, no process in $\mathcal{S}$ calls advance in v_1. On the other hand, by Claim 1 some process in $\mathcal{S}$ enters $v_1 + 1$. Then by Validity, some process in $\mathcal{S}$ calls advance in v_1, which is a contradiction. ◀

6 Possibility and Impossibility of Classical Consensus via Ω

We now present some interesting consequences of our results. It is well-known that the failure detector Ω [16, 17] is sufficient for implementing wait-free consensus resilient to $\lfloor \frac{n-1}{2} \rfloor$ failures in non-partitionable systems under crash [17, 35], crash/recovery [5], or general omission [35, 48] failures. We now investigate if this is still the case under flaky channels and limited connectivity. Below we demonstrate that the answer depends on the amount of connectivity in the underlying network, as established by our lower and upper bounds.

Lower bound. We first show that, on the negative side, there are some weakly synchronous environments where Ω is implementable, but obstruction-free consensus is impossible even if at most one process can crash. One such environment is the system S of Aguilera et al. [3], where in every execution, all channels can be flaky except those emanating from an a priori unknown correct process (*timely source*); the latter channels are required to be eventually reliable and timely. In our framework, S is represented by the set of executions in $\mathcal{M}_{\text{PEF}}$ compliant with the following fail-prone system:

$$\mathcal{F}_S = \{(P, C) \mid |P| < n \wedge \exists p \in \mathcal{P} \setminus P.\ \forall q \in \mathcal{P} \setminus (P \cup \{p\}).\ (p, q) \notin C\}.$$

We now use Theorem 10 to prove that consensus is impossible even in a stronger variant of S where only up to a threshold k of processes are allowed to fail. Formally, for n and k such that $0 < k < n$, let $\mathcal{F}_{S,k} = \mathcal{F}_S \cap \{(P, C) \mid |P| \leq k\}$. Then $\mathcal{F}_{S,k}$ is a k-fail-prone system.

▶ **Theorem 15.** *Let n and k be such that $0 < k < n$, and $\tau : \mathcal{F}_{S,k} \to 2^{\mathcal{P}}$ be a termination mapping such that $\forall f \in \mathcal{F}_{S,k}.\ \tau(f) \neq \emptyset$. Then no algorithm can implement an $(\mathcal{F}_{S,k}, \tau)$-obstruction-free consensus in $\mathcal{M}_{\text{PEF}}$.*

Proof. By contradiction, assume such an algorithm exists. First, a usual partitioning argument similar to that of Dwork et al. [23] can be used to show that we must have $n > 2$ and $k \leq \lfloor \frac{n-1}{2} \rfloor$. Then, since $\mathcal{M}_{\text{PRD}}$ is stronger than $\mathcal{M}_{\text{PEF}}$, Theorem 10 requires that for all $f \in \mathcal{F}_{S,k}$, $\mathcal{G} \setminus f$ must contain a strongly connected component of size $> \lfloor \frac{n-1}{2} \rfloor \geq 1$. However, for a failure pattern $f = (P, C) \in \mathcal{F}_{S,k}$ such that

$$|P| = \emptyset \wedge \exists p \in \mathcal{P}.\ \forall q \in \mathcal{P} \setminus (P \cup \{p\}).\ \forall r \in \mathcal{P} \setminus P.\ (q, r) \in C,$$

all strongly connected components of $\mathcal{G} \setminus f$ are of size 1: a contradiction. ◀

Upper bounds. On the positive side, we now show that strengthening connectivity assumptions does enable consensus to be solved in many previously proposed models of weak synchrony, such as systems S^+ and S^{++} of Aguilera et al. [3] and those defined in [2,24,32,38]. These papers show that Ω can be implemented in their respective models, but (with an exception of [2]) do not give a consensus algorithm.

To show that consensus is indeed possible in these models, we introduce an intermediate model $\mathcal{S}^\star$ in which all channels can be asynchronous and flaky apart from those connecting an a priori unknown correct process (a *hub* [3]) to all the other processes in *both* directions; the latter channels are required to be eventually reliable (but can still be asynchronous). In our framework, $\mathcal{S}^\star$ is represented by the set of executions of $\mathcal{M}_{\text{AEF}}$ compliant with the following fail-prone system:

$$\mathcal{F}_{\mathcal{S}^\star} = \{(P, C) \mid |P| < n \wedge \exists p \in \mathcal{P} \setminus P.\ \forall q \in \mathcal{P} \setminus (P \cup \{p\}).\ (p, q) \notin C \wedge (q, p) \notin C\}.$$

Thus, for all $f \in \mathcal{F}_{\mathcal{S}^\star}$, $\mathcal{G} \setminus f$ is strongly connected, and in the case when at most $\lfloor \frac{n-1}{2} \rfloor$ processes fail in f, $\mathcal{G} \setminus f$ is a connected core. Then from Theorem 8, we get

▶ **Corollary 16.** *It is possible to implement a wait-free atomic register in $\mathcal{S}^\star$ if at most $\lfloor \frac{n-1}{2} \rfloor$ processes can crash.*

Since wait-free consensus tolerating any number of $< n$ process crashes can be implemented using atomic registers and Ω [36], Corollary 16 implies

▶ **Corollary 17.** *It is possible to implement a wait-free consensus in $\mathcal{S}^\star$ augmented with Ω if at most $\lfloor \frac{n-1}{2} \rfloor$ processes can crash.*

We now use this result to prove

▶ **Corollary 18.** *It is possible to implement wait-free consensus over the systems S^+ and S^{++} of [3] and the models of [2, 24, 32, 38], provided at most $\lfloor \frac{n-1}{2} \rfloor$ processes can crash.*

Proof. We first show that the systems enumerated in the theorem's statement are as strong as $\mathcal{S}^\star$. System S^+ [3] stipulates that every execution has a *fair hub*, i.e., a correct process which is connected to all other processes via fair-lossy channels [1] in both directions. Since eventually reliable channels can be built on top of fair-lossy ones in an asynchronous system, S^+ is as strong as $\mathcal{S}^\star$. In S^{++} [3] and [2], every pair of processes is assumed to be connected by fair-lossy channels. Thus, similarly to the above, S^{++} is as strong as $\mathcal{S}^\star$. Finally, the models of [24,32,38] assume that every pair of processes is connected via reliable channels, which again means that these models are strictly stronger than $\mathcal{S}^\star$. Since Ω can be implemented in all models mentioned above [2,3,24,32,38], Corollary 17 implies the required. ◀

7 Related Work

Fault-tolerant distributed computing in the presence of message loss has been extensively studied in the past. Early work focused on the models that, in addition to crashes, allow processes to also fail to either send messages (*send omission*) [28] or both send and receive messages (*generalized omission*) [47]. Both these models have been shown to be equivalent to the crash failures in their computational power in both synchronous [43] and asynchronous [19] systems. They are however too strong to capture partitionable systems, as they would automatically classify any non-crashed process with unreliable connectivity as faulty.

The classical paper by Dolev [20] as well as more recent work (see [49] for survey) study consensus solvability in general networks as a function of the network topology, process failure models, and synchrony constraints. These papers however, only consider the classical (i.e., non-partitionable) version of consensus, which requires agreement to be reached by all correct processes. They also do not consider failure models with flaky channels.

In a seminal paper [50], Santoro and Widmayer proposed a *mobile omission* failure model, which treats message loss independently of process failures. They showed that in order to solve consensus in a synchronous round-based system in the absence of process failures, it is necessary and sufficient to ensure that the communication graph contains a strongly connected component in every round [50, 51]. In contrast, our lower bounds demonstrate that in order to implement consensus (or even a register) in a partially synchronous system, some processes must remain strongly connected throughout the *entire* execution.

Subsequent work explored fault-tolerant computation in the presence of both faulty links and processes under weakened synchrony assumptions. Basu et al. [11] established a separation between reliable channels and either eventually reliable or fair-lossy channels in terms of their power to solve certain problems (such as uniform reliable broadcast and k-set agreement) under asynchrony. However, their notion of a reliable link [29] is too strong to be implementable in practice as it requires every message transmitted via a complete send invocation to be eventually delivered to a correct destination even if the sender later crashes.

Several generalizations of the classical failure detector abstractions of [17] for partitionable systems were proposed in [4,21,22,26] and shown sufficient for consensus in [4,21,26]. However, as we explain in §1, the failure detectors introduced in these papers cannot be implemented in the presence of flaky channels.

Aguilera et al. [3] and follow-up work [2, 24, 32, 38] studied the problem of implementing a failure detector Ω (which is the weakest for consensus [16]), under various weak models of synchrony, link reliability, and connectivity. However, these results are inapplicable in our setting as the models considered in these papers disallow full partitions.

As we show in §6, our connectivity bounds for consensus are also applicable in non-partitionable systems with unreliable channels. In particular, they imply that Ω is not the only factor determining consensus solvability, and that the degree of connectivity being assumed also plays an important role. This suggests an interesting tradeoff across all three modeling dimensions of synchrony, channel reliability, and connectivity considered in this paper. In particular, to solve consensus we assume that every non-faulty channel is eventually timely [23], whereas the algorithm of [2] can cope with weaker synchrony constraints. However, unlike [2], which assumes complete connectivity via fair-lossy channels, our implementation is able to tolerate an adversarial message loss (flakiness). Whether this tradeoff is fundamental, or our synchrony assumptions can be further relaxed remains the subject of future work.

Alquraan et al. [7] present a study of system failures due to network partitions, which we already mentioned in §1. This work highlights the practical importance of coming up with provably correct designs that explicitly consider channel failures. A follow-up paper [6]

introduces a communication layer that can mask channel failures, but only when access to low-level networking infrastructure is available. OmniPaxos [44] ensures liveness of consensus under channel failures, but only handles disconnected channels, not flaky ones.

Raft [45,46] elects leaders in a randomized way: a prospective leader picks a view number and requests *Votes* from a majority of processes (analogous to 1A messages of Paxos). This may lead to split votes when multiple processes are competing to get elected, in which case each process waits for a randomized timeout and retries. While this mechanism works reasonably well in practice, it does not guarantee termination under partial synchrony even with perfect connectivity. Raft also includes a *Pre-Vote* optimization, which, as Jensen et al. point out [33], can help maintain liveness under intermittent connectivity. The optimization requires that before sending *Vote* requests, a prospective leader gathers a majority of *Pre-Votes* from other processes, certifying that they are ready to vote for it. This is similar to gathering WISHes from a majority of processes in our synchronizer before entering a view (Figure 4). But unlike our synchronizer *Pre-Vote* is a best-effort technique, since in between a process granting a *Pre-Vote* and receiving the corresponding *Vote* request it may vote for someone else, thereby creating a split vote. Nevertheless, the presence of techniques similar to ours in existing practical algorithms makes us hopeful that our work could be used to make Paxos-based systems provably live even under adversarial network conditions.

References

1 Yehuda Afek, Hagit Attiya, Alan D. Fekete, Michael J. Fischer, Nancy A. Lynch, Yishay Mansour, Da-Wei Wang, and Lenore D. Zuck. Reliable communication over unreliable channels. *J. ACM*, 41(6):1267–1297, 1994. doi:10.1145/195613.195651.

2 Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Symposium on Principles of Distributed Computing (PODC)*, 2004. doi:10.1145/1011767.1011816.

3 Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing Omega in systems with weak reliability and synchrony assumptions. *Distributed Comput.*, 21(4):285–314, 2008. doi:10.1007/S00446-008-0068-Y.

4 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theor. Comput. Sci.*, 220(1):3–30, 1999. doi:10.1016/S0304-3975(98)00235-7.

5 Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Comput.*, 13(2):99–125, 2000. doi:10.1007/S004460050070.

6 Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2020. URL: https://www.usenix.org/conference/osdi20/presentation/alfatafta.

7 Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2018. URL: https://www.usenix.org/conference/osdi18/presentation/alquraan.

8 Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, 1995. doi:10.1145/200836.200869.

9 Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. Lower bound on the step complexity of anonymous binary consensus. In *Symposium on Distributed Computing (DISC)*, 2016. doi:10.1007/978-3-662-53426-7_19.

10 Peter Bailis and Kyle Kingsbury. The network is reliable. *Commun. ACM*, 57(9):48–55, 2014. doi:10.1145/2643130.

11 Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Workshop on Distributed Algorithms (WDAG)*, 1996. doi:10.1007/3-540-61769-8_8.

12 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Liveness and latency of Byzantine state-machine replication. In *Symposium on Distributed Computing (DISC)*, 2022. doi:10.4230/LIPICS.DISC.2022.12.

13 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine consensus live. *Distributed Comput.*, 35(6):503–532, 2022. doi:10.1007/S00446-022-00432-Y.

14 Eric A. Brewer. Towards robust distributed systems (abstract). In *Symposium on Principles of Distributed Computing (PODC)*, 2000. doi:10.1145/343477.343502.

15 Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/brooker.

16 Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996. doi:10.1145/234533.234549.

17 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996. doi:10.1145/226643.226647.

18 Gregory Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001. doi:10.1145/503112.503113.

19 Brian Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Trans. Comput.*, 37(12):1541–1553, 1988. doi:10.1109/12.9732.

20 Danny Dolev. The Byzantine generals strike again. *J. Algorithms*, 3(1):14–30, 1982. doi:10.1016/0196-6774(82)90004-9.

21 Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. Technical Report TR96-1608, Department of Computer Science, Cornell University, 1996.

22 Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments (brief announcement). In *Symposium on Principles of Distributed Computing (PODC)*, 1997.

23 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988. doi:10.1145/42282.42283.

24 Antonio Fernández Anta and Michel Raynal. From an intermittent rotating star to a leader. In *Conference on Principles of Distributed Systems (OPODIS)*, 2007. doi:10.1007/978-3-540-77096-1_14.

25 Faith Fich, Maurice Herlihy, and Nir Shavit. On the space complexity of randomized synchronization. *J. ACM*, 45(5):843–862, 1998. doi:10.1145/290179.290183.

26 Roy Friedman, Idit Keidar, Dahlia Malkhi, Ken Birman, and Danny Dolev. Deciding in partitionable networks. Technical Report TR95-1554, Department of Computer Science, Cornell University, 1995.

27 Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. doi:10.1145/564585.564601.

28 Vassos Hadzilacos. 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, 1983.

29 Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcast and related problems. Technical Report TR94-1425, Department of Computer Science, Cornell University, 1994.

30 Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, 1991. doi:10.1145/114005.102808.

31 Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *International Conference on Distributed Computing Systems (ICDCS)*, 2003. doi:10.1109/ICDCS.2003.1203503.

32 Martin Hutle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou. Chasing the weakest system model for implementing ω and consensus. *IEEE Trans. Dependable Secur. Comput.*, 6(4):269–281, 2009. doi:10.1109/TDSC.2008.24.

33 Chris Jensen, Heidi Howard, and Richard Mortier. Examining Raft's behaviour during partial network failures. In *Workshop on High Availability and Observability of Cloud Systems (HAOC)*, 2021. doi:10.1145/3447851.3458739.

34 Leslie Lamport. On interprocess communication – Part I: Basic formalism, Part II: Algorithms. *Distributed Comput.*, 1(2):77–101, 1986. doi:10.1007/BF01786227.

35 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998. doi:10.1145/279227.279229.

36 Wai-Kau Lo and Vassos Hadzilacos. Using failure detectors to solve consensus in asynchronous shared-memory systems (extended abstract). In *Workshop on Distributed Algorithms (WDAG)*, 1994. doi:10.1007/BFb0020440.

37 Nancy Lynch. *Distributed Algorithms*, chapter 17. Morgan Kaufmann, 1996.

38 Dahlia Malkhi, Florin Oprea, and Lidong Zhou. ω meets paxos: Leader election and stability without eventual timely links. In *Symposium on Distributed Computing (DISC)*, 2005. doi:10.1007/11561927_16.

39 Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Comput.*, 11(4):203–213, 1998. doi:10.1007/S004460050050.

40 Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. Cogsworth: Byzantine view synchronization. In *Cryptoeconomics Systems Conference (CES)*, 2020. doi:10.21428/58320208.08912a03.

41 Oded Naor and Idit Keidar. Expected linear round synchronization: The missing link for linear Byzantine SMR. In *Symposium on Distributed Computing (DISC)*, 2020. doi:10.4230/LIPICS.DISC.2020.26.

42 Alejandro Naser-Pastoriza, Gregory Chockler, and Alexey Gotsman. Fault-tolerant computing with unreliable channels (extended version), 2023. arXiv:2305.15150.

43 Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *J. Algorithms*, 11(3):374–419, 1990. doi:10.1016/0196-6774(90)90019-B.

44 Harald Ng, Seif Haridi, and Paris Carbone. Omni-Paxos: Breaking the barriers of partial connectivity. In *European Conference on Computer Systems (EuroSys)*, 2023. doi:10.1145/3552326.3587441.

45 Diego Ongaro. *Consensus: bridging theory and practice*. PhD thesis, Stanford University, USA, 2014. URL: https://searchworks.stanford.edu/view/10608105.

46 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, 2014. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

47 Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Software Eng.*, 12(3):477–482, 1986. doi:10.1109/TSE.1986.6312888.

48 Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the PAXOS algorithm. *Theor. Comput. Sci.*, 243(1-2):35–91, 2000. doi:10.1016/S0304-3975(00)00042-6.

49 Dimitris Sakavalas and Lewis Tseng. *Network Topology and Fault-Tolerant Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2019. doi:10.2200/S00918ED1V01Y201904DCT016.

50 Nicola Santoro and Peter Widmayer. Time is not a healer. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, 1989. doi:10.1007/BFB0028994.

51 Nicola Santoro and Peter Widmayer. Distributed function evaluation in the presence of transmission faults. In *Symposium on Algorithms (SIGAL)*, 1990. doi:10.1007/3-540-52921-7_85.