UNIVERSIDAD POLITÉCNICA DE MADRID
Escuela Técnica Superior de Ingenieros Informáticos

Fault-Tolerant Computing with Unreliable Channels

DOCTORAL THESIS

Submitted for the degree of Doctor by:

Alejandro Naser-Pastoriza
Licenciado en Ciencias de la Computación

Madrid, 2025

UNIVERSIDAD POLITÉCNICA DE MADRID
Escuela Técnica Superior de Ingenieros Informáticos

Doctoral Degree in Software, Systems and Computing

Fault-Tolerant Computing with Unreliable Channels

DOCTORAL THESIS

Submitted for the degree of Doctor by:

Alejandro Naser-Pastoriza
Licenciado en Ciencias de la Computación

Under the supervision of:
Dr. Alexey Gotsman

Madrid, 2025

Title: Fault-Tolerant Computing with Unreliable Channels

Author: Alejandro Naser-Pastoriza

Doctoral Programme: Software, Systems and Computing

Thesis Supervision:

Dr. Alexey Gotsman, Research Professor, IMDEA Software Institute (Supervisor)

External Reviewers:

Thesis Defense Committee:

Thesis Defense Date:

A Noelia Sofía.

Acknowledgements

I would like to express my deepest gratitude to the people who have supported and guided me throughout my PhD journey.

I am especially thankful to my advisor, Alexey Gotsman, and my collaborator, Gregory Chockler, for their invaluable feedback, collaboration, and countless insightful conversations over the course of this work. Their guidance has greatly enriched the quality and depth of my research.

I would also like to thank my managers during my internships at Amazon Web Services – Byron Cook, Ernie Cohen, and Mark Tuttle – for their mentorship and for giving me the opportunity to work on challenging projects. These experiences have played an important role in shaping my perspective and growth as a researcher.

I am grateful to everyone at the IMDEA Software Institute for providing a supportive environment during my PhD. I want to especially thank my friends, Fedor Ryabinin and Margarita Capretto, for their companionship and encouragement throughout this journey.

Last, and most importantly, I am profoundly grateful to my wife, Noelia Ledesma Lopez, for her constant encouragement, patience, and unwavering support.

To all of you – thank you for being part of my PhD.

Abstract

This thesis studies the foundations of fault-tolerant distributed computing under unreliable communication. Motivated by the practical challenges of message loss in modern networks, we consider a general model where both processes and communication channels may fail. Unlike classical models, we allow channels to be flaky, meaning they can lose an arbitrary number of messages without fairness or eventual delivery guarantees. This setting reflects a wide range of real-world failures observed in modern distributed applications.

Our first contribution is a general characterization of solvability under arbitrary combinations of process and channel failures. We introduce a framework called a generalized quorum system (GQS), which precisely captures the minimal connectivity needed to implement classical shared-memory abstractions such as atomic registers, atomic snapshots, lattice agreement, and consensus. We prove that a GQS is both necessary and sufficient for solving these problems, even under extremely weak connectivity conditions. In particular, we show that these abstractions can be implemented even when read quorums are not strongly connected, as long as some strongly connected write quorum is reachable from them.

To match this lower bound, we develop algorithms for each of the four problems in an adversarial model where correct channels are only eventually reliable and faulty channels are flaky. A key technical challenge is implementing registers under these assumptions. To address this, we develop two new building blocks: quorum access functions and custom logical clocks, which enable indirect coordination between weakly connected quorums.

We then shift our attention to a practically motivated class of fail-prone systems in which at most a minority of processes may crash. This assumption, although restrictive, allows stronger guarantees and more efficient algorithms. In this setting, we show that consensus becomes the most difficult abstraction to implement efficiently. Classical mechanisms like leader election or failure detectors are not effective under flaky communication. To overcome this, we adapt a recently proposed abstraction of a view synchronizer, which enables groups of correct processes to coordinate in a common view with a correct leader. Using this abstraction, we design a modular consensus protocol that guarantees wait-freedom within a connected core of the system, using bounded memory and without assuming transitive connectivity.

Finally, we demonstrate how our bounds can be used to analyze consensus solvability in existing models of weak connectivity. In particular, we show that some models strong enough to implement a leader detector are still too weak to solve consensus.

Overall, this thesis contributes new theoretical foundations, abstractions, and algorithms for reliable distributed computing under weak connectivity. It provides a unified framework for reasoning about solvability and complexity in settings that better reflect the behavior of modern networks.

Resumen

Esta tesis estudia los fundamentos de la computación distribuida tolerante a fallos en presencia de comunicación no confiable. Motivados por los desafíos prácticos que impone la pérdida de mensajes en redes modernas, consideramos un modelo general donde tanto los procesos como los canales de comunicación pueden fallar. A diferencia de los modelos clásicos, permitimos canales *flaky*, que pueden perder una cantidad arbitraria de mensajes sin garantías de equidad ni entrega eventual. Este escenario refleja una amplia gama de fallos reales observados en aplicaciones distribuidas actuales.

Nuestra primera contribución es una caracterización general de la posibilidad de implementación bajo combinaciones arbitrarias de fallos de procesos y canales. Introducimos una nueva herramienta, el *sistema de quórums generalizado* (GQS, por sus siglas en inglés), que captura de forma precisa la conectividad mínima necesaria para implementar abstracciones clásicas de memoria compartida, como registros atómicos, instantáneas atómicas, acuerdo sobre retículos y consenso. Demostramos que un GQS es una condición necesaria y suficiente para resolver estos problemas, incluso bajo supuestos de conectividad extremadamente débiles. En particular, mostramos que estas abstracciones pueden implementarse incluso si los quórums de lectura no son fuertemente conexos, siempre que exista un quórum de escritura fuertemente conexo alcanzable desde ellos.

Para igualar esta cota inferior, desarrollamos algoritmos para cada uno de los cuatro problemas bajo un modelo adversario donde los canales correctos son solo eventualmente fiables y los canales defectuosos son *flaky*. Un desafío clave es la implementación de registros en este contexto. Para resolverlo, desarrollamos dos mecanismos: funciones de acceso a quórum y relojes lógicos personalizados, que permiten la coordinación indirecta entre quórums débilmente conectados.

Luego analizamos una clase de sistemas motivada por la práctica, en los que como mucho puede fallar una minoría de procesos. Aunque esta suposición restringe el conjunto de fallos posibles, permite ofrecer garantías más fuertes y algoritmos más eficientes. En este caso, demostramos que consenso se vuelve la abstracción más difícil de implementar de manera eficiente. Mecanismos clásicos como detectores de fallos o detectores de líder no son eficaces bajo canales *flaky*. Para superar esta dificultad, adaptamos una abstracción propuesta recientemente llamada sincronizador de vistas, que permite a grupos de procesos correctos coordinarse en una vista común con un líder correcto. Con esta abstracción, diseñamos un protocolo modular de consenso que garantiza progreso en el núcleo conectado del sistema, utiliza memoria acotada y no asume conectividad transitiva.

Finalmente, mostramos cómo nuestras cotas pueden aplicarse al análisis de modelos existentes con conectividad débil. En particular, demostramos que algunos modelos suficientemente fuertes para implementar detectores de líder no alcanzan para resolver consenso.

En conjunto, esta tesis ofrece nuevas bases teóricas, abstracciones y algoritmos para la computación distribuida confiable bajo conectividad parcial. Proporciona un marco unificado para razonar sobre posibilidad de implementación y complejidad en entornos que reflejan mejor el comportamiento de las redes modernas.

Table of Contents

List of Figures

Abbreviations and acronyms

ABD	Attiya, Bar-Noy, and Dolev (register algorithm)
CAP	Consistency, Availability, Partition-tolerance (CAP theorem)
FLP	Fischer, Lynch, and Paterson (impossibility result)
FIFO	First In, First Out
GST	Global Stabilization Time
GQS	Generalized Quorum System
MWMR	Multi-Writer Multi-Reader (register or snapshot)
OPODIS	International Conference on Principles of Distributed Systems
PODC	ACM Symposium on Principles of Distributed Computing
QS	Quorum System
SWMR	Single-Writer Multi-Reader (register or snapshot)
Ω	Eventual Leader Detector
$\diamond P$	Eventually Perfect Failure Detector

Chapter 1

Introduction

Tolerating communication channel failures is an important challenge faced by the developers of modern distributed systems. This issue is of practical significance, as such failures frequently occur in real-world deployments [15, 20, 44]. They stem from a range of causes – including physical infrastructure breakdowns, software bugs in network switches, and configuration errors – and often lead to severe system outages. In fact, Alquraan et al. [10] conducted a comprehensive study of failures caused by network partitions in widely used replicated data stores. Their findings showed that the majority of these failures had catastrophic consequences, and that resolving nearly half required redesigning core system mechanisms. Thus, these failures were not merely the result of programming errors, but reflected deeper flaws in system design.

Perhaps the most challenging aspect of real-world network failures is that networks can break down in arbitrarily complex ways, resulting in a wide variety of connectivity patterns:

- In the simplest case, individual channel failures do not affect the overall network connectivity (Figure 1.1a).
- Sometimes, the network partitions into multiple components with no connectivity between them (Figure 1.1b).
- In more complex scenarios, some components remain connected in one direction but not in the other (Figure 1.1c).
- In the worst case, some components become intermittently connected, leading to the loss of an arbitrary subset of messages transmitted between them (Figure 1.1d).

The latter pattern has so far received little attention in theoretical research. Most network models for classical problems in distributed computing assume that channels are either (eventually) reliable or (eventually) disconnected (Figures 1.1a – c). Reliable channels are sometimes replaced by *fair lossy* ones, which guarantee delivery only if a message is 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 is acknowledged and by filtering out duplicates [3].

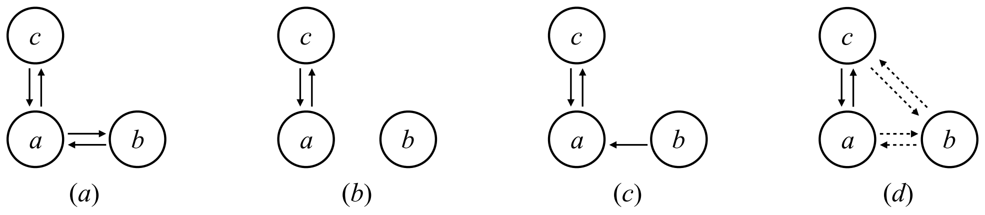

Figure 1.1: Examples of irregular connectivity patterns: (a) indirect connectivity; (b) clean partitions; (c) asymmetric connectivity; and (d) intermittent connectivity. All processes are correct. Solid arrows indicate reliable channels, dashed arrows denote channels that may arbitrarily drop messages, and missing arrows represent disconnected channels.

What makes channel failures even more difficult to handle is that they must be tolerated in conjunction with ordinary process failures, often leading to complex and subtle interactions. 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 $n \geq 2k + 1$ [12, 32]. Subsequent work [42, 50] generalized this result to the case where 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 may crash in a single execution [52]. In this setting, the problems mentioned above are implementable under a given fail-prone system if and only 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 thesis, we extend these results to fail-prone systems that may include arbitrary combinations of both process and channel failures – namely, process crashes and channels that may arbitrarily drop messages.

Example 1.1. Consider a set of processes $\mathcal{P} = \{a, b, c, d\}$. Figure 1.2 depicts a fail-prone system $\mathcal{F}$ consisting of failure patterns f_i, $i = 1..4$ (ignore R_i and W_i for now). Under failure pattern f_1, processes a, b, and c are correct, while d may crash. Channels (c, a), (a, b), and (b, a) are correct, while all others may fail.

A plausible conjecture for a tight reliability bound under fail-prone systems comprising arbitrary combinations of process and channel failures might be formulated as follows. There exists 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 [12] and Paxos [47]. In fact, QS$^+$ was shown to be sufficient for partially synchronous consensus [7, 30, 31, 37].

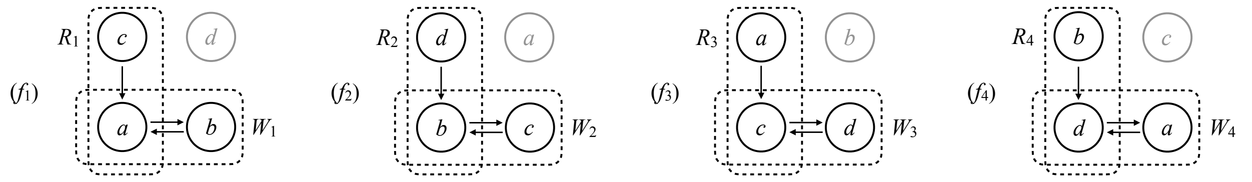

Figure 1.2: A fail-prone system $\mathcal{F}$ and a generalized quorum system $(\mathcal{F}, \mathcal{R}, \mathcal{W})$, where $\mathcal{R} = \{R_i \mid i = 1..4\}$ and $\mathcal{W} = \{W_i \mid i = 1..4\}$. Solid circles denote correct processes, gray circles denote processes that may crash, solid arrows denote reliable channels, and missing arrows denote channels that may fail.

These results, however, leave a noticeable gap. While the existence of QS$^+$ is sufficient in some cases, 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 [52, 53]. Surprisingly, we answer this question in the negative. More broadly, this thesis addresses the following research questions:

- **RQ1:** How can existing theoretical models be generalized to precisely capture arbitrary combinations of process and channel failures, including the failure patterns observed in practical distributed systems?
- **RQ2:** What are the minimal reliability conditions on processes and communication channels under which classical fault-tolerant abstractions – such as atomic registers, atomic snapshots, lattice agreement, and consensus – can be solved?
- **RQ3:** What novel abstractions, algorithms, and techniques are necessary to efficiently implement these fault-tolerant primitives under adversarial conditions, including arbitrary process crashes and unconstrained message loss?

By addressing these questions, this thesis provides a unified theoretical foundation and practical methodologies for designing robust distributed systems that tolerate both process and channel failures. Our results consist of lower and upper bounds that characterize the fundamental limits of reliability under such failures (§3.5).

Lower bounds. We first present a general framework that extends the classical notion of a *fail-prone system* [52] to account for both process and channel failures (§3.2). This framework allows us to express, for example, whether an algorithm is intended to tolerate the failure patterns illustrated in Figure 1.1. We then use this framework to establish minimal connectivity requirements necessary to implement atomic registers, atomic snapshots, lattice agreement, and consensus.

To obtain strong lower bounds, we consider strong network models in which correct channels are reliable and faulty channels are disconnected, i.e., drop all messages. We assume asynchrony for registers, snapshots, and lattice agreement (model $\mathcal{M}_{\text{ARD}}$ in Figure 1.3), and partial synchrony [32] for consensus (model $\mathcal{M}_{\text{PRD}}$): executions begin asynchronously and eventually become synchronous. The most notable aspect of our lower bounds is that, unlike prior work under unreliable connectivity (e.g., [5, 61]), they are proven under a weak termination guarantee that only requires obstruction-free termination at a *subset* of the processes (§3.3).

	Reliable / Disconnected (lower bound)	Eventually Reliable / Flaky (upper bound)
Asynchronous (registers, snapshots, lattice agreement)	$\mathcal{M}_{\text{ARD}}$	$\mathcal{M}_{\text{AEF}}$
Partially synchronous (consensus)	$\mathcal{M}_{\text{PRD}}$	$\mathcal{M}_{\text{PEF}}$

Figure 1.3: System models categorized by synchrony assumptions and channel types.

Informally, we establish two necessary conditions for implementing the problems listed above (§4.1 and §5.1):

1. All processes where obstruction-freedom holds must be strongly connected via correct channels.
2. There must be sufficient reliable connectivity for some correct write quorum to be both strongly connected and *unidirectionally* reachable from some correct read quorum – a condition we formalize through a novel *generalized quorum system* (GQS) (§3.4).

We prove that, given a fail-prone system $\mathcal{F}$ comprising an arbitrary set of process-channel failure patterns, if atomic registers, atomic snapshots, lattice agreement, or consensus are implementable under $\mathcal{F}$, then $\mathcal{F}$ admits a GQS.

Example 1.2. Consider the fail-prone system $\mathcal{F} = \{f_i \mid i = 1..4\}$ in Figure 1.2. 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 W_i is strongly connected and reachable from R_i via channels correct under the failure pattern f_i. None of the read quorums R_i are strongly connected, thus relaxing the connectivity requirements of QS^+.

Our results also show that any solution to the problems above can guarantee termination only within the write quorums of some GQS (e.g., processes a and b under failure pattern f_1 from Figure 1.2). Such a restricted termination guarantee is expected in our setting, as channel failures may isolate some correct processes, making it impossible to ensure termination for all of them.

The above results generalize the celebrated CAP theorem [19, 38], which states that it is impossible to guarantee all three of Consistency, Availability, and network Partition-tolerance (§4.2). Whereas CAP asserts only that *some* channels must be correct to ensure consistency and availability, we identify *which* ones are needed. Furthermore, while CAP requires availability at *all* processes, we characterize the conditions under which it can be ensured in just *a part* of the system.

Upper bounds. Next, we propose algorithms for atomic registers, atomic snapshots, lattice agreement, and consensus that match our lower bounds (§4.3 and §5.2). These algorithms assume the existence of a GQS and guarantee wait-freedom at its write quorums. They are designed to operate in an adversarial model where correct channels are only *eventually* reliable and faulty channels are *flaky*, i.e., they may drop an arbitrary subset of messages sent on them, without any guarantee of fairness (models $\mathcal{M}_{\text{AEF}}$ and $\mathcal{M}_{\text{PEF}}$ in Figure 1.3).

Flakiness is a broad failure mode that captures a range of patterns observed in practice, including both full and intermittent loss of connectivity. It also subsumes previously studied variants of lossy channels, such as eventually disconnected [7, 25] and fair lossy [3, 16]. Although irregular connectivity patterns have been investigated before [6, 7, 30, 31, 37], to the best of our knowledge, we are the first to propose implementations of registers, snapshots, lattice agreement, and consensus in the presence of flaky channels.

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

Example 1.3. Assume that a register operation is invoked at process a under failure pattern f_1 from Figure 1.2. In this case, all channels incoming to 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. However, because the network is asynchronous, it is difficult for a to determine whether the information received from c is up to date – that is, whether it reflects all updates that precede the current operation invocation at a, a critical requirement for ensuring linearizability.

We address this challenge using a novel logical clock that processes use to tag the information they push downstream. The clock is 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. Since snapshots and lattice agreement can be reduced to registers, the upper bounds for those problems follow as well.

Next, we show that the existence of a GQS also constitutes a tight bound on the connectivity required to implement consensus under partial synchrony. Interestingly, we find that 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 whether the information it receives is up to date.

Bounded process failures. The upper bounds presented earlier allow us to show that our bounds are tight, thereby settling the computability question. However, the corresponding algorithms are inneficient in terms of both latency and message complexity. To address this issue, we shift our attention to a practically motivated class of fail-prone systems in which at most a minority of processes may fail. While this assumption restricts the set of admissible failure patterns, it yields stronger connectivity guarantees (§4.4.1 and §5.3) that can be leveraged to design more efficient implementations – specifically, implementations in which the time required to complete an operation can be bounded in terms of message delays and the rate at which processes exchange state (§4.4.2 and §5.4).

Informally, we show that for any n-process implementation of the problems mentioned before, if the implementation tolerates k process crashes and $n = 2k + 1$, then any process where obstruction-freedom holds must belong to a set of at least $k + 1$ correct processes that are strongly connected by correct channels. We call the largest such set the *connected core* of the network. For example, in Figure 1.1a, the connected core is $\{a, b, c\}$, whereas in Figures 1.1b–d it is $\{a, c\}$.

The implementation of consensus is the more challenging one. Although classical abstractions such as failure detectors or leader election mechanisms [24] can be adapted to solve consensus in models with disconnections or fair-lossy channels [7, 25, 30, 37, 61], they are no longer effective in the presence of flaky channels. Intuitively, this is because they may fail to correctly identify processes with reliable connectivity – a requirement for ensuring the liveness of consensus.

Example 1.4. Suppose that the pattern of message loss experienced by the channels (a, b), (b, a), (b, c), and (c, b) in Figure 1.1d is such that all leader election messages are delivered. Thus, it is possible for process b to be elected as a leader. However, since any message sent by process b after being elected can be dropped – regardless of how many times it is resent – the process may not be able to drive consensus to completion, thereby violating liveness.

To overcome the limitations of failure and leader detectors under flaky channels, we take a different approach: we generalize the abstraction of a *view synchronizer* [17, 18, 54, 55], recently proposed for Byzantine consensus, to benign failure settings with flaky channels. Roughly, a view synchronizer enables a commonly used design pattern in which the protocol execution is divided into *views* (also known as *rounds*), each with a designated leader that coordinates process interactions within that view.

The task of the synchronizer is to ensure that a sufficiently large set of correct processes eventually spends enough time in the same view with a correct leader, who can then drive consensus to completion. Supporting this functionality under the model $\mathcal{M}_{\text{PEF}}$ is nontrivial, as during asynchronous periods clocks can diverge and messages used to synchronize views may be lost or delayed. View synchronizers encapsulate the logic required to handle these challenges, thereby enabling the 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 argued above, is difficult with flaky channels), but instead delegates this responsibility to the top-level protocol. The protocol can monitor the current leader using timeouts and other protocol-specific logic, and request the synchronizer to switch to another view with a different leader if it detects a lack of progress. For example, if processes a and c in Figure 1.1d do not observe progress from process b due to message loss, they would eventually initiate a view change and try a different leader.

We present a specification of a view synchronizer sufficient to implement consensus in the presence of flaky channels, along with an implementation and a proof that the implementation satisfies this specification (§5.4.1, §5.4.2 and §5.4.3). While handling crashes is easier than Byzantine faults, flaky channels introduce a distinct challenge: processes outside the connected core (such as process b in Figures 1.1b–d) may falsely suspect the current leader and should not be able to force a view change, thereby disrupting a functioning view. Using our synchronizer, we then design a consensus protocol that tolerates flaky channels and prove its correctness.

Finally, we demonstrate how our lower and upper bounds can be applied to establish consensus solvability in various existing models of weak connectivity [5, 6, 33, 43, 51] (§5.5). In particular, we show that some connectivity models strong enough to implement the Ω leader detector are nevertheless too weak to implement consensus.

To conclude, we observe that all upper bounds under bounded process failures are achieved using bounded-memory algorithms, reflecting the practical performance motivations underlying this part of the thesis.

The results presented in this thesis are based on the following peer-reviewed publications:

- **Alejandro Naser-Pastoriza**, Gregory Chockler, Alexey Gotsman, Fedor Ryabinin.
  Tight Bounds on Channel Reliability via Generalized Quorum Systems.
  In **Proceedings of the 44th ACM Symposium on Principles of Distributed Computing (PODC 2025)**, pages 444–454. Editors: Fabian Kuhn (Program Chair). Published by ACM. CORE A*.
  Available at: https://doi.org/10.1145/3732772.3733529.
- **Alejandro Naser-Pastoriza**, Gregory Chockler, Alexey Gotsman.
  Fault-Tolerant Computing with Unreliable Channels. **Best Paper Award**.
  In **Proceedings of the 27th International Conference on Principles of Distributed Systems (OPODIS 2023)**, volume 286 of LIPIcs, pages 21:1–21:21. Editors: Alysson Bessani, Xavier Défago, Junya Nakamura, Koichi Wada, Yukiko Yamauchi. Published by Schloss Dagstuhl – Leibniz-Zentrum für Informatik. CORE B.
  Available at: https://doi.org/10.4230/LIPIcs.OPODIS.2023.21.

Chapter 2

Background

This chapter provides the background needed to make the thesis self-contained. It introduces the system model, correctness conditions, core concepts, and classical abstractions that are used throughout the thesis.

We begin by recalling that modern distributed systems are typically composed of a group of processes. A process abstracts a unit of computation – such as a physical or virtual machine, a processor, or a thread of execution in a concurrent environment. A key goal in designing such systems is to enable these processes to cooperate on a shared task.

The environment – whether it's a single computer, a data center, or a system spread across the globe – introduces several challenges. One of them is to design mechanisms that enable reliable cooperation, even in the presence of process or network failures, or under adversarial conditions. The central difficulty in distributed computing is to build algorithms that let working processes continue collaborating correctly on the common task, despite partial failures.

To manage the complexity of distributed systems, recurring patterns of cooperation are often encapsulated in well-defined abstractions. These act as modular building blocks upon which more complex systems can be designed. In this thesis, we focus on four such abstractions: atomic registers, atomic snapshots, lattice agreement, and consensus.

Before we define these abstractions formally, we first introduce the computational model in which they are expressed. This includes identifying the main elements of a distributed system – processes, communication channels, timing assumptions, and failures – and abstracting their behavior to avoid low-level implementation details. This formalization helps us reason precisely about the correctness and complexity of distributed algorithms.

2.1 Basic Definitions

Processes. A unit capable of performing computation in a distributed system is modeled as a *process*. We assume the system consists of a fixed set of processes $\mathcal{P}$, known to all participants. Each process runs its own algorithm; together, these local programs define a *distributed algorithm*.

```
1 when received ACCEPT($b, k, m$) from $p_j$
2     pre: status $\in$ {LEADER, FOLLOWER} $\wedge$ bal $= b$
3     log[$k$] $= m$
4     send ACCEPT_ACK($b, k$) to $p_j$
```

Figure 2.1: Example of a handler.

When a process deviates from the prescribed algorithm, we say that a *failure* has occurred. The types of failures allowed define the process abstraction. In this thesis, we assume the *crash-stop* model, in which a process may halt and never resume execution.

Channels. We consider message-passing systems, where processes communicate by exchanging messages over *channels* that abstract the underlying physical network. Channels may lose, duplicate, reorder, or arbitrarily delay messages, but they do not corrupt them or generate spurious ones.

Timing. It is also important to specify the system's timing assumptions – such as whether processes have access to clocks or whether communication delays are bounded. In this thesis, we consider both *asynchronous* systems, which make no such assumptions, and models with *partial synchrony*. The precise assumptions will be introduced later, as needed.

System Model. A distributed system model is defined by combining the abstractions introduced above. Reasoning about distributed algorithms begins by fixing the model in which they are intended to operate. This model then serves as a foundation for introducing higher-level abstractions that capture recurring patterns of interaction in distributed applications.

2.2 Pseudocode

We describe our algorithms using pseudocode that follows a reactive model, where each algorithm consists of a set of event handlers. Each handler specifies how a process responds to a particular type of event.

In this asynchronous, event-driven model, each process behaves as a state machine whose transitions are triggered by events: either the reception of one or more messages or the satisfaction of conditions on local variables. A handler is written using a **when** block, followed by a sequence of instructions, as illustrated in Figure 2.1.

Each process handles events atomically and one at a time – no two events are processed concurrently by the same process. The only exception is when a handler is suspended on an **await** statement, in which case other handlers may execute until the condition of **await** is satisfied. After completing one event, a process checks whether another event has been triggered. This check is assumed to be fair and is implicit – it is not shown in the pseudocode.

Handlers may also be guarded by preconditions on the local state (line 2 in Figure 2.1). A handler executes only when its event is triggered and its condition is satisfied. We assume the runtime system buffers such events until their conditions hold.

2.3 Correctness Conditions

The correctness of a distributed algorithm is typically defined in terms of *safety* and *liveness* properties. A safety property ensures that nothing bad ever happens, while a liveness property guarantees that something good eventually happens. We describe these two classes of properties below.

A *safety property* asserts that an algorithm never violates a certain invariant. If a violation does occur, it must happen in finite time, and the property cannot be restored afterward. Formally, a property is a safety property if, whenever it is violated in an execution σ, there exists a finite prefix σ' such that every extension of σ' also violates it.

A *liveness property*, in contrast, asserts that some desired event will eventually occur. Such properties cannot be violated in finite time: for any finite execution, it remains possible that the property will be satisfied later. Formally, a property is a liveness property if every finite execution σ can be extended to an infinite execution σ' that satisfies it.

A correct implementation of a shared object must also guarantee that operations eventually complete for some correct processes. These guarantees are formalized through *progress conditions*, which are a subclass of liveness properties. *Obstruction-freedom* holds at a process if every operation it invokes eventually completes, provided the process runs alone for long enough without interference from others. *Wait-freedom* holds at a process if every operation it invokes eventually completes, regardless of the behavior or failures of other processes.

2.4 Fail-Prone Systems

In classical models of distributed computing, failure assumptions are often cardinality-based – for example, assuming that at most k out of $2k + 1$ processes may crash. A more general and modular approach is to explicitly specify the failure scenarios that an algorithm must tolerate. This is formalized through the notion of a *fail-prone system* [52].

A *failure pattern* captures a single failure scenario, describing which processes may fail in an execution. Formally, a failure pattern is a set $f \subseteq \mathcal{P}$. A *fail-prone system* captures all failure scenarios that an algorithm must tolerate. Formally, a fail-prone system is a collection $\mathcal{F} \subseteq 2^{\mathcal{P}}$ of possible failure patterns. An algorithm is said to tolerate $\mathcal{F}$ if it satisfies its correctness properties in every execution where the set of faulty processes is contained in some $f \in \mathcal{F}$.

For example, the classical threshold failure assumption – *at most k process failures* – corresponds to the fail-prone system:

$$\mathcal{F} = \{f \subseteq \mathcal{P} \mid |f| \leq k\}.$$

Fail-prone systems offer a flexible way to describe complex failure models. They allow capturing constraints that go beyond thresholds – for example, that certain processes fail together, or that some are always correct. This level of generality is especially useful when reasoning about fault-tolerant algorithms for non-uniform environments.

2.5 Quorum Systems

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

Classical quorum systems are defined under the assumption that only processes may fail, and that channels between correct processes are reliable. In this setting, failure assumptions are captured by a *fail-prone system* [52], as defined in the previous section.

Definition 2.1. *A* ***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 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*, i.e.,* $R \cup W \subseteq \mathcal{P} \setminus f$*.*

Example 2.2. Consider a system with n processes, where up to $k \leq \lfloor \frac{n-1}{2} \rfloor$ may fail. Let:

- $\mathcal{F} = \{f \subseteq \mathcal{P} \mid |f| \leq k\}$ – up to k process failures.
- $\mathcal{R} = \{R \subseteq \mathcal{P} \mid |R| \geq n - k\}$ – read quorums of size at least $n - k$.
- $\mathcal{W} = \{W \subseteq \mathcal{P} \mid |W| \geq k + 1\}$ – write quorums of size at least $k + 1$.

Then $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ is a quorum system with respect to $\mathcal{F}$ [12, 42].

This example illustrates the usefulness of distinguishing between read and write quorums: it allows smaller write quorums in exchange for larger read quorums. In the special case where $k = \lfloor \frac{n-1}{2} \rfloor$, we get $\mathcal{R} = \mathcal{W}$, so both read and write quorums are majorities.

2.6 Core Objects: Specifications and Implementations

This section introduces the shared objects that are the focus of this thesis: multi-writer multi-reader (MWMR) atomic registers, single-writer multi-reader (SWMR) atomic snapshots, lattice agreement, and consensus. These abstractions capture common cooperation patterns in distributed computing and are used extensively in the design of fault-tolerant distributed systems.

We begin by formalizing the safety properties that define the correct behavior of read/write objects. Then, for each abstraction, we describe a classical implementation under the assumption of asynchronous reliable channels and crash-stop processes. These implementations rely on the quorum system described in Example 2.2, tolerate up to k process crashes, and guarantee wait-freedom at all correct processes. Together, these baseline algorithms serve as a reference point for the fault-tolerant constructions we present later in the thesis.

2.6.1 Common Definitions

An operation op' *follows* an operation op, denoted $op \to op'$, if op' is invoked after op returns; op' is *concurrent* with op if neither $op \to op'$ nor $op' \to op$. An execution is *sequential* if no two operations in it are concurrent.

An execution σ of an object (e.g., a register or snapshot) is *linearizable* [40] if there exists a set of responses R 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 R), such that π is a correct sequential execution and satisfies $op_i \to op_j \implies i < j$.

An execution σ 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 of an object is *safe* [46] if all its executions satisfy safeness; it is *atomic* if all its executions are linearizable.

2.6.2 MWMR Atomic Registers

Specification. Let Value be an arbitrary domain of values. A *multi-writer multi-reader atomic register* supports two operations: write(x), which stores a value $x \in$ Value and returns *ack*; and read, which retrieves the current value from the register and returns it. A sequential execution of these operations is *correct* if every read operation returns the value written by the most recent preceding write, or a special value $\perp \in$ Value if there is no such preceding write.

Implementation. In Figure 2.2, we present the implementation of a multi-writer multi-reader (MWMR) variant of the single-writer multi-reader (SWMR) atomic register implementation originally proposed by Attiya, Bar-Noy, and Dolev [12, 50].

The implementation tags values with versions from $\mathbb{N} \times \mathbb{N}$, ordered lexicographically. A version is a pair consisting of a monotonically increasing number and a process identifier. Each register process maintains a local state $(\mathsf{val}, \mathsf{ver})$, where val is the most recent value written to the register at this process, and ver is its associated version. Each process also maintains a monotonically increasing sequence number seq, which is used to generate a unique identifier for each register operation invoked at the process. This identifier is included in all messages exchanged during the operation, enabling the process to associate incoming responses with the corresponding invocation.

To execute a write(x) operation (line 4), a process follows two phases:

- **Get phase.** The process increments its sequence number (line 5) and broadcasts `GET_REQ`(seq) to all processes (line 6). Upon receiving a `GET_REQ`(s) message (line 20), each process replies with its current value and version by sending `GET_RESP`$(\mathsf{val}, \mathsf{ver}, s)$ (line 21). The writer waits to collect such replies from a read quorum R (line 7). It then selects the maximum version $(k, \_)$ among the responses and constructs a new version $(k + 1, i)$ using its own identifier i (line 8). This ensures that the new version is strictly greater than any previously observed, preserving a consistent and increasing order across writes.

```
1  seq ← $0$
2  val ← $\perp$
3  ver ← $(0, i)$
4  function write($x$):
5      seq ← seq $+ 1$
6      send GET_REQ(seq) to all
7      wait until received $\{$GET_RESP$(val_j, ver_j,$ seq$) \mid p_j \in R\}$
           from some read quorum $R$
8      $(k, \_) \leftarrow \max\{ver_j \mid p_j \in R\}$
9      send SET_REQ$(x, (k+1, i),$ seq$)$ to all
10     wait until received $\{$SET_RESP(seq)$ \mid p_j \in W\}$
           from some write quorum $W$
11     return $ack$
12 function read():
13     seq ← seq $+ 1$
14     send GET_REQ(seq) to all
15     wait until received $\{$GET_RESP$(val_j, ver_j,$ seq$) \mid p_j \in R\}$
           from some read quorum $R$
16     let $p_k \in R$ be such that $\forall p_j \in R.\ ver_k \geq ver_j$
17     send SET_REQ$(val_k, ver_k,$ seq$)$ to all
18     wait until received $\{$SET_RESP(seq)$ \mid p_j \in W\}$
           from some write quorum $W$
19     return $val_k$
20 when received GET_REQ$(s)$ from $p_j$
21     send GET_RESP(val, ver, $s$) to $p_j$
22 when received SET_REQ$(val, ver, s)$ from $p_j$
23     if $ver \geq$ ver then
24         val ← $val$
25         ver ← $ver$
26     send SET_RESP$(s)$ to $p_j$
```

Figure 2.2: MWMR atomic register protocol at a process p_i.

- **Set phase.** The process then stores the pair $(x, (k + 1, i))$ at some write quorum. To this end, it sends SET_REQ$(x, (k + 1, i),$ seq$)$ to all processes (line 9). Upon receiving such a message (line 22), a process updates its local value and version if the received version is greater than or equal to its current one (lines 23–25), and then replies with a SET_RESP(seq) message (line 26). The writer waits to collect acknowledgments from a write quorum W (line 10) and then returns ack (line 11). This ensures that the value x is stored at a sufficiently large set of processes to be retrievable by subsequent read operations.

To execute a read() operation (line 12), a process follows two similar phases:

- **Get phase.** The process increments its sequence number (line 13) and sends `GET_REQ`(seq) to all processes (line 14). As before, each process responds with its local value and version (lines 20–21). The reader waits to collect `GET_RESP`$(val_j, ver_j, \mathsf{seq})$ messages from a read quorum R (line 15) and selects the pair (val_k, ver_k) such that ver_k is the maximum among the received versions (line 16). This ensures that the reader observes a value that reflects the most recent completed write, or a concurrent one that has already reached a write quorum.

- **Set phase.** To ensure that the read value remains visible to future operations, the reader performs a write-back. This step is necessary to propagate the observed value to a write quorum, preventing it from being overwritten or lost due to concurrent or incomplete writes. The reader sends `SET_REQ`$(val_k, ver_k, \mathsf{seq})$ to all processes (line 17). Each recipient compares the received version with its local one, updates its state if the received version is greater or equal (lines 22–25), and replies with `SET_RESP`(seq) (line 26). The reader waits for responses from a write quorum W (line 18) and then returns val_k (line 19). Like in the Set phase of a write operation, this guarantees that the value val_k is stored at a write quorum and can be retrieved by subsequent read operations.

2.6.3 SWMR Atomic Snapshots

Specification. Let Value be an arbitrary domain of values. A *single-writer multi-reader atomic snapshot* object consists of a collection of *segments*, where each segment holds a value in Value. In the single-writer setting, each process is assigned a unique segment that only it can write to, while all processes may read from any segment. The object supports two operations: write(x), which stores a value $x \in \mathsf{Value}$ in the process's own segment and returns *ack*; and read, which retrieves the values of all segments as a *vector* over Value. A sequential execution is *correct* if every read operation returns a vector that reflects the most recent preceding write to each segment, or a special value $\perp \in \mathsf{Value}$ if there is no such preceding write.

Implementation. In Figure 2.3, we present the SWMR atomic snapshot implementation by Afek et al. [2]. Our description of the algorithm and its behavior borrows explanations and intuition from their original presentation.

The construction builds on the abstraction of atomic registers introduced earlier, used as black-box shared objects. Each process p_i owns a register r_i, which it writes and all processes read. Each register stores three fields: a value $\mathsf{val}(r_i) \in \mathsf{Value}$, initially $\perp$; a sequence number $\mathsf{seq}(r_i) \in \mathbb{Z}$, initially 0; and a view $\mathsf{view}(r_i) \in \mathsf{Value}^n$, initially $\perp^n$. Each process also maintains a local boolean vector $\mathsf{moved}[1 \ldots n]$, initially all zero, used to track whether any process appears to have performed a write during a read.

The helper operation collect reads the current contents of all n registers in arbitrary order and returns a vector of triples $(\mathsf{val}(r_j), \mathsf{seq}(r_j), \mathsf{view}(r_j))$ for $j = 1 \ldots n$. These represent the most recent value, sequence number, and view stored by each process.

The snapshot algorithm relies on two fundamental insights:

```
1  seq ← 0
2  function read():
3      for $j = 1 \ldots n$ do
4          moved[$j$] ← 0
5      let $a$ = collect
6      let $b$ = collect
7      if $\forall j \in \{1 \ldots n\}$. seq($a_j$) = seq($b_j$) then    // nobody moved
8          return (val($b_1$), ..., val($b_n$))
9      for $j = 1 \ldots n$ do
10         if seq($a_j$) ≠ seq($b_j$) then    // $p_j$ moved
11             if moved[$j$] then    // $p_j$ moved once before!
12                 return view($b_j$)
13             else
14                 moved[$j$] ← 1
15     go to line 5
16 function write($x$):
17     seq ← seq + 1
18     let $s$ = read()
19     $r_i$.write($x$, seq, $s$)
```

Figure 2.3: SWMR atomic snapshot protocol at a process p_i.

- **Observation 1.** If two consecutive collects return identical results, and the second begins after the first ends, then the returned values form a consistent snapshot.
- **Observation 2.** If a read observes a process perform two distinct writes (i.e., its sequence number changes twice), then that process must have completed a write operation during the read.

To execute a read() operation (line 2), a process repeatedly collects the state of the system until it can return a consistent snapshot. It begins by initializing the moved vector to all zeros (line 3). It then performs two consecutive collects: the first into vector a (line 5) and the second into vector b (line 6). If $\mathsf{seq}(a_j) = \mathsf{seq}(b_j)$ for all $j \in \{1, \ldots, n\}$, then the reader returns the vector $(\mathsf{val}(b_1), \ldots, \mathsf{val}(b_n))$ as the snapshot result (line 8), by Observation 1.

If some process p_j has changed its sequence number – i.e., $\mathsf{seq}(a_j) \neq \mathsf{seq}(b_j)$ – the reader records that p_j has moved (line 9). If this is the second time p_j is observed to move (i.e., $\mathsf{moved}[j] = 1$), then, by Observation 2, p_j must have completed a write during the read. In that case, the reader returns the view stored in p_j's register, i.e., $\mathsf{view}(b_j)$ (line 12). Otherwise, the reader marks $\mathsf{moved}[j] = 1$ and restarts the double collect process (line 15).

This loop guarantees termination. In each iteration, the reader either returns a snapshot, or marks a process as having moved. Therefore, if it has not returned earlier, after at most $n + 1$ iterations some process must be observed to have moved twice, triggering a return via $\mathsf{view}(b_j)$ (line 12). This ensures that the read operation is wait-free.

```
1 let $S$ be a shared atomic snapshot object
2 function propose($x$):
3     $S$.write($x$)
4     let $s = S$.read()
5     return $\bigsqcup s$
```

Figure 2.4: Lattice agreement protocol at a process p_i.

To execute a write(x) operation (line 16), a process first increments its local sequence number (line 17), then invokes read() (line 18) to obtain a vector $s \in \mathsf{Value}^n$ representing a consistent snapshot of the system. It then writes the triple (x, seq, s) to its own register r_i (line 19). This view enables future readers to reuse the writer's snapshot if they observe that the write completed during their own read, thereby ensuring consistency even in the presence of concurrent updates.

2.6.4 Lattice Agreement

Specification. In the lattice agreement problem, each process p_i may invoke an operation 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 $\bigsqcup$. 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 $y_i \leq \bigsqcup X$, where X is the set of x_j for which propose(x_j) was invoked.

Implementation. In Figure 2.4, we present a simple implementation of lattice agreement that builds on the abstraction of atomic snapshots introduced earlier, used as a black-box shared object. Each process p_i accesses the snapshot object S through its dedicated segment, using the operations write and read.

To execute propose(x) (line 2), a process writes its input value x to its segment of S (line 3), then invokes S.read() to capture a snapshot of the system state (line 4). This yields a vector s containing the latest values written by all processes. The process returns the join of the values in s under the lattice's $\bigsqcup$ operator (line 5).

The algorithm satisfies the properties of lattice agreement. For Downward validity, each process reads after writing its input x_i, so x_i must appear in the snapshot and thus $x_i \leq y_i$. For Upward validity, the snapshot contains only values written by processes that invoked propose(x_j), so $y_i \leq \bigsqcup X$, where X is the set of all such x_j. For Comparability, note that each process writes exactly once to its segment, and the snapshot object is atomic. Hence, any two snapshots reflect sets of inputs where one is included in the other. Since joins are monotonic under inclusion, the outputs must be comparable in the lattice.

2.6.5 Consensus

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

Implementation. In Figure 2.5, we present an implementation of consensus in the style of Flexible Paxos [42]. The algorithm is designed to guarantee the safety properties of consensus – Agreement and Validity. In the next section, we introduce a mechanism to ensure termination, thereby satisfying the liveness requirement.

We begin with a high-level overview of the protocol and a description of the local state maintained by each process. The protocol follows a leader-driven approach. To propose a value, a process must first be elected as leader. A leader wins an election once it receives votes from a read quorum, thereby obtaining the right to propose values. To this end, the process stores the value it wishes to propose in the variable my_val and broadcasts an invitation for other processes to join its ballot and vote for it as leader.

Once elected, the process enters a period of execution called a *ballot*. At any time, each process participates in a unique ballot, stored in the variable bal. A control variable phase tracks the current stage of the protocol: ELECTION, PROPOSE, ACCEPT, or DECIDE.

To reach a decision, the leader must convince every member of a write quorum to accept its proposal. To this end, it broadcasts the proposal to all processes. Upon receiving it, a follower acknowledges the message, thereby accepting the proposal. Each process persists its most recently accepted value in the variable val and records the ballot in which it was accepted in cbal. Once a write quorum has accepted the same value in the same ballot, the proposal is considered agreed and can be safely decided.

We now describe the protocol in more detail. The execution proceeds in two phases. In the first phase, a process attempts to gather support from a read quorum to participate in its current ballot. During this phase, it collects information about any values that may have already been accepted in earlier ballots by other processes. This information is essential to preserve consistency across proposals and to ensure that previously chosen values are not inadvertently overwritten.

To propose a value x, process p_i invokes propose(x) (line 4). It sets my_val $= x$, chooses a ballot number $b >$ bal such that leader(b) $= p_i$, and initiates a leader election by broadcasting a message `1A`(b) to all processes (line 6). This message expresses its intent to lead ballot b and marks the beginning of the ELECTION phase.

Upon receiving a message `1A`(b) (line 9), a process checks whether $b >$ bal. If so, it joins ballot b by updating bal and entering the ELECTION phase. It replies with `1B`(b, cbal, val) (line 13), reporting the most recently accepted value and the ballot in which it was accepted.

The leader waits to collect `1B` messages from a read quorum R. If all reports indicate that no value has been previously accepted (i.e., $x_j = \perp$ for all $p_j \in R$), the leader is free to propose

```
1  $\mathsf{bal}, \mathsf{cbal} \leftarrow 0, 0$
2  $\mathsf{val}, \mathsf{my\_val} \leftarrow \bot, \bot$
3  $\mathsf{phase} \leftarrow$ ELECTION
4  **function** $\mathsf{propose}(x)$:
5      $\mathsf{my\_val} \leftarrow x$
6      **send** $\mathtt{1A}(b \mid b > \mathsf{bal} \land \mathsf{leader}(b) = p_i)$ **to all**
7      **async wait until** $\mathsf{phase} =$ DECIDE
8      **return** $\mathsf{val}$
9  **when received** $\mathtt{1A}(b)$ **from** $p_j$
10     **pre:** $b > \mathsf{bal}$
11     $\mathsf{bal} \leftarrow b$
12     $\mathsf{phase} \leftarrow$ ELECTION
13     **send** $\mathtt{1B}(b, \mathsf{cbal}, \mathsf{val})$ **to** $p_j$
14 **when received** $\{\mathtt{1B}(\mathsf{bal}, cbal_j, x_j) \mid p_j \in R\}$ **from a read quorum** $R$
15     **pre:** $\mathsf{phase} =$ ELECTION
16     **if** $\forall p_j \in R.\ x_j = \bot$ **then**
17         **if** $\mathsf{my\_val} = \bot$ **then return**
18         **send** $\mathtt{2A}(\mathsf{bal}, \mathsf{my\_val})$ **to all**
19     **else**
20         **let** $p_j \in R$ **be such that** $x_j \neq \bot$ **and**
21             $(\forall p_k \in R.\ x_k \neq \bot \Rightarrow cbal_k \leq cbal_j)$
22         **send** $\mathtt{2A}(\mathsf{bal}, x_j)$ **to all**
23     $\mathsf{phase} \leftarrow$ PROPOSE
24 **when received** $\mathtt{2A}(b, x)$
25     **pre:** $\mathsf{bal} \leq b$
26     $\mathsf{val} \leftarrow x$
27     $\mathsf{bal}, \mathsf{cbal} \leftarrow b, b$
28     **send** $\mathtt{2B}(b, x)$ **to all**
29     $\mathsf{phase} \leftarrow$ ACCEPT
30 **when received** $\{\mathtt{2B}(\mathsf{bal}, x) \mid p_j \in W\}$ **from a write quorum** $W$
31     $\mathsf{val} \leftarrow x$
32     $\mathsf{cbal} \leftarrow \mathsf{bal}$
33     $\mathsf{phase} \leftarrow$ DECIDE
```

Figure 2.5: Consensus protocol at a process p_i.

its own value my_val (line 16). Otherwise, to preserve agreement, it selects the value x_j that was accepted in the highest reported ballot $cbal_j$ (line 21).

In the second phase, the leader attempts to get a write quorum to accept the proposed value v. To this end, it broadcasts a message $\mathtt{2A}(\mathsf{bal}, v)$ to all processes and transitions to the PROPOSE phase (line 23).

When a process receives $\mathtt{2A}(b, v)$ (line 24), it checks that $b \geq \mathsf{bal}$ and, if so, updates bal, sets $\mathsf{cbal} = b$ and $\mathsf{val} = v$, and broadcasts an acknowledgment $\mathtt{2B}(b, v)$ to all processes (line 28). The process then transitions to the ACCEPT phase.

Finally, if the leader or any other process receives $\mathtt{2B}(b, v)$ acknowledgments from a write quorum W (line 30), it concludes that the value v has been accepted by a write quorum. It sets $\mathsf{val} = v$, records $\mathsf{cbal} = \mathsf{bal} = b$, and transitions to the DECIDE phase (line 33). This transition satisfies the guard checked by the proposer in line 7, allowing the invocation of $\mathsf{propose}(x)$ to return val to the application.

The protocol guarantees that only one value can be decided across all ballots. This can be proved based on the intersection of read and write quorums, together with the requirement that leaders adopt the highest accepted value previously reported.

2.7 Failure Detectors

It is well established that termination of consensus cannot be guaranteed in asynchronous systems subject to crash failures [36]. The core challenge lies in the inability to distinguish a process that has crashed from one that is merely slow. To circumvent this limitation, consensus protocols must rely on some degree of synchrony.

These synchrony assumptions are needed only to enable failure detection – the ability to eventually suspect crashed processes. As a result, they are typically captured through failure detectors: modules that provide possibly unreliable information about which processes may have crashed. A failure detector can be seen as an oracle that each process periodically queries for hints about which processes are still alive. Therefore, when used as black-box components, failure detectors make it possible to solve consensus in an otherwise asynchronous system.

The weakest failure detector that suffices for solving consensus under crash failures is Ω [24]. It provides an eventual leader election mechanism: for an arbitrary period of time, leadership may be unstable – leaders may change, multiple leaders may coexist, and some may crash. However, Ω guarantees that eventually all correct processes agree on the same correct process, which remains the leader forever. This enables termination by providing a stable and correct process capable of eventually driving the protocol to completion. In this section, we present its formal specification and provide a simple implementation under partial synchrony [32].

To implement Ω, we rely on the *Eventually Perfect Failure Detector* $\diamond P$, which provides each process with information about which other processes are believed to have crashed. This information may be inaccurate for an arbitrary period, but eventually all faulty processes are permanently suspected, and all correct ones are permanently trusted. At a high level, the detector operates by exchanging periodic heartbeat messages and progressively increasing timeouts to tolerate delays – this is where synchrony is required. Each process uses this information to select the highest-ranked process not suspected as its current leader. Eventually, all correct processes make the same choice, thereby implementing Ω.

We then revisit the consensus implementation from Figure 2.5. We modularize it by decoupling the leader election mechanism – based on Ω – from the core consensus logic. Our presentation builds on ideas from Algorithm 5.5 in [21], but the original algorithm contains a subtle bug that breaks correctness under certain message interleavings. We illustrate this issue and explain how to fix it.

2.7.1 Eventually Perfect Failure Detector $\diamond P$

Specification. The eventually perfect failure detector $\diamond P$ provides two upcalls: suspect(p) and restore(p), where $p \in \mathcal{P}$. The call suspect(p) indicates that process p is believed to have crashed, whereas restore(p) revokes that suspicion. A process may be suspected and later restored an arbitrary number of times. The detector must satisfy the following properties: eventually, every process that crashes is permanently suspected by every correct process (Strong completeness); and eventually, no correct process is suspected by any correct process (Eventual strong accuracy).

Implementation. Figure 2.6 presents the implementation of $\diamond P$ due to Cachin et al. [21]. Our description of the algorithm and its behavior borrows intuition from their original presentation. The algorithm assumes partial synchrony [32] (see §3.1) and that all channels are reliable.

The protocol uses periodic heartbeat exchanges to monitor the liveness of other processes. Each process p_i maintains two local sets: alive, which tracks the processes that responded in the current round, and suspected, which contains the processes currently considered faulty. Initially, alive includes all processes (line 1), suspected is empty (line 3), and a delay parameter delay is initialized to a constant $C > 0$ (line 2). This parameter defines the timeout between consecutive heartbeat rounds.

On startup, p_i sets a timer to expire after delay time units (line 5). When the timer expires, p_i first checks whether any process appears in both alive and suspected (line 7). This condition indicates a false suspicion: the process was previously suspected but responded during this round. In that case, delay is increased by C (line 8), allowing more time for heartbeat responses in future rounds. This adaptive mechanism ensures that delay eventually grows large enough to accommodate network delay between correct processes, thereby satisfying Eventual strong accuracy.

Next, p_i iterates over all processes in $\mathcal{P}$ (line 9). If a process p has not responded during the round, i.e., $p \notin$ alive, and was not previously suspected, then p is added to suspected (line 11) and the upcall suspect(p) is triggered (line 12). Conversely, if p did respond during the round, i.e., $p \in$ alive, and had previously been suspected, then it is removed from suspected (line 14) and restore(p) is triggered (line 15). Regardless of their status, all processes are sent a heartbeat request `HEARTBEAT_REQUEST` in every round (line 16). After the loop, the set alive is cleared in preparation for the next round (line 17), and the timer is restarted using the current value of delay (line 18).

When p_i receives a `HEARTBEAT_REQUEST` message from some p_j, it replies immediately with a `HEARTBEAT_RESPONSE` message (line 20). Upon receiving `HEARTBEAT_RESPONSE` from p_j, p_i records p_j in alive (line 22), meaning it was responsive during this round. As soon as a crashed process stops replying, it is no longer added to alive and will eventually be suspected and never restored, ensuring Strong completeness.

```
alive ← $\mathcal{P}$
delay ← $C$
suspected ← $\emptyset$

on startup
    start_timer(heartbeat_timer, delay)

when the timer heartbeat_timer expires
    if alive $\cap$ suspected $\neq \emptyset$ then
        delay ← delay + $C$
    for $p \in \mathcal{P}$ do
        if $p \notin$ alive $\wedge$ $p \notin$ suspected then
            suspected ← suspected $\cup \{p\}$
            trigger suspect($p$)
        else if $p \in$ alive $\wedge$ $p \in$ suspected then
            suspected ← suspected $\setminus \{p\}$
            trigger restore($p$)
        send HEARTBEAT_REQUEST to $p$
    alive ← $\emptyset$
    start_timer(heartbeat_timer, delay)

when received HEARTBEAT_REQUEST from $p_j$
    send HEARTBEAT_RESPONSE to $p_j$

when received HEARTBEAT_RESPONSE from $p_j$
    alive ← alive $\cup \{p_j\}$
```

Figure 2.6: $\diamond P$ protocol at a process p_i. C is a positive constant.

2.7.2 Eventual Leader Detector Ω

Specification. The eventual leader detector Ω provides a single upcall, trust(p), where $p \in \mathcal{P}$, indicating that process p is currently trusted as the leader. Once this happens, p is considered trusted until a subsequent upcall names a different process. The detector must satisfy the following properties: eventually, every correct process trusts some correct process (Accuracy); and eventually, no two correct processes trust different correct processes (Agreement).

Implementation. In Figure 2.7, we present an implementation of Ω that builds on the eventually perfect failure detector $\diamond P$, used as a black-box shared object. Each process p_i has access to $\diamond P$ using the upcalls suspect and restore.

The implementation maintains a local variable suspected, initialized to the empty set (line 2), which stores the processes that p_i currently believes to have crashed. The output of the failure detector is stored in the variable leader (line 1), which identifies the process currently trusted as leader by p_i. Initially, this value is $\perp$.

When $\diamond P$ notifies p_i via the upcall suspect(p), the process adds p to its suspected set (line 4), indicating that p is currently believed to have crashed. Similarly, upon receiving restore(p),

```
1 leader $\leftarrow \bot$
2 suspected $\leftarrow \emptyset$
3 on suspect($p$)
4     suspected $\leftarrow$ suspected $\cup \{p\}$
5 on restore($p$)
6     suspected $\leftarrow$ suspected $\setminus \{p\}$
7 when leader $\neq \max(\mathcal{P} \setminus$ suspected)
8     leader $\leftarrow \max(\mathcal{P} \setminus$ suspected)
9     trigger trust(leader)
```

Figure 2.7: Ω protocol at a process p_i.

the process removes p from suspected (line 6), reflecting that p is now considered alive. This ensures that the suspected set always reflects p_i's current view of which processes are faulty, as determined by the underlying $\diamond P$ detector.

Whenever leader differs from the process of highest identifier among those not in suspected (i.e., $\max(\mathcal{P} \setminus \text{suspected})$), p_i updates leader accordingly (line 8) and triggers the upcall trust(leader) (line 9). This mechanism guarantees that the identity of the currently trusted leader is always aligned with the most up-to-date view of suspected processes.

The use of $\max(\mathcal{P} \setminus \text{suspected})$ ensures that processes eventually converge toward trusting the same leader. Since $\diamond P$ satisfies Strong completeness, every process that crashes is eventually and permanently added to suspected. Moreover, Eventual strong accuracy guarantees that eventually no correct process is suspected by any correct process. As a result, there is a time after which $\mathcal{P} \setminus$ suspected contains only correct processes, and all correct processes agree on the same maximum element. From that point onward, every correct process outputs the same correct leader forever, thereby satisfying both Accuracy and Agreement.

2.8 Consensus Termination via Ω

The liveness property of consensus requires that every propose invocation at a correct process eventually terminates. While the protocol in Figure 2.5 is safe under all execution scenarios, it does not guarantee this property: there is no assurance that processes will remain in a given ballot for sufficiently long to allow the process leading that ballot to drive the protocol to completion.

We show how to ensure the liveness of consensus by relying on the Ω eventual leader detector (see §2.7.2), used as a black box in an otherwise asynchronous system with reliable channels. In particular, we examine a protocol by Cachin et al. [21] that introduces an Ω-based ballot change mechanism designed to eventually establish a stable and correct leader. This leader should then be able to propose and stabilize a ballot that is not rejected by any correct process, enabling consensus to terminate. These requirements are formally specified in [21, Module 5.3].

```
 1 bal ← 1
 2 next_bal ← $i$
 3 trusted ← $p_1$
 4 on trust($p$)
 5     if $p \neq$ trusted then
 6         send NACK to trusted
 7     trusted ← $p$
 8     if $p = p_i$ then
 9         next_bal ← next_bal + $|\mathcal{P}|$
10         send 1A(next_bal) to all
11 when received 1A($b$) from $p_j$
12     if trusted $= p_j \wedge b >$ bal then
13         bal ← $b$
14     else
15         send NACK to $p_j$
16 when received NACK
17     if trusted $= p_i$ then
18         next_bal ← next_bal + $|\mathcal{P}|$
19         send 1A(next_bal) to all
```

Figure 2.8: Ballot change mechanism from [21, Algorithm 5.5].

As we will demonstrate, however, the implementation by Cachin et al. fails to meet these guarantees and does not ensure liveness as intended. We describe the protocol, illustrate the flaw, and propose a fix. This case highlights the difficulty of ensuring liveness: the original implementation in the book [21] contained a bug, later corrected in errata [22]; yet even with this fix, the protocol may still fail to terminate. Hence, it provides a valuable point of comparison for our solutions in models with channel failures.

Implementation. In Figure 2.8, we present the protocol by Cachin et al. [21, Algorithm 5.5] incorporating published errata [22] and written in our style. The protocol builds on the Ω leader detector, used as a black-box shared object. Each process p_i has access to Ω through the trust upcall.

Each process p_i maintains three local variables: next_bal, the highest ballot number that p_i has attempted to lead (initialized to i in line 2); bal, the highest ballot in which p_i has participated (initialized to 1, line 1); and trusted, which stores the identifier of the currently trusted leader (initialized to p_1, line 3). These initial values imply that all processes initially consider p_1 to be the leader in ballot 1, and have already participated in that ballot.

Whenever Ω outputs trust(p), the handler checks whether this represents a change in leadership. If so, it notifies the previously trusted process by sending it a NACK (line 6). This serves to inform the old leader that it is no longer considered the leader by the current process. The handler then updates trusted (line 7). If the newly trusted process is p_i itself, it increments next_bal by $|\mathcal{P}|$ (line 9) to ensure ballot uniqueness, and broadcasts a 1A message with the updated ballot number (line 10).

Upon receiving a message 1A(b) from a process p_j, a process checks whether p_j is currently trusted and whether b exceeds its local bal. If both conditions hold, it accepts the ballot and updates bal (line 13). Otherwise, it rejects the message and replies with a NACK (line 15).

Intuitively, the latter NACK serves to notify the prospective leader that the sender is already in a higher ballot. This indicates that the leader should retry with a larger ballot number in order to eventually reach and be accepted by all correct processes. To this end, if a leader receives a NACK while still trusted, it increments next_bal and broadcasts a new 1A message (lines 18–19).

A counterexample to Eventual Leadership [21, Module 5.3]. The ballot change mechanism requires that, eventually, all correct processes start a final ballot, initiate no further ones, and agree on the same final ballot led by a correct process. This property, known as Eventual Leadership, is not satisfied by the implementation in Figure 2.8. We demonstrate this by constructing an execution in which ballots grow unboundedly, even after Ω has stabilized.

Let $\mathcal{P} = \{p_1, p_2, p_3\}$, and assume all processes are correct. Initially, trusted $= p_1$ at every process. Suppose that Ω triggers trust(p_1) at p_1 and p_3 at the beginning of the execution and never triggers another upcall at those processes. At p_1, this satisfies the guard in line 8, so it sets next_bal $= 4$ and broadcasts 1A(4). At p_3, since it does not trust itself, the guard fails and no action is taken.

Next, Ω triggers trust(p_3) at p_2, followed by trust(p_1). The first upcall causes p_2 to send a NACK to p_1 and update trusted $= p_3$; the second causes it to send a NACK to p_3 and update trusted $= p_1$. Since p_2 never trusts itself, the guard in line 8 is never satisfied, so it does not initiate any ballots. No further trust upcalls occur at any process, so Ω has stabilized on p_1.

Deliver the NACK to p_3 first. Since p_3 does not trust itself, the guard in line 17 fails and the message is ignored. Then deliver the NACK to p_1. Since p_1 trusts itself, it increments next_bal to 7 and broadcasts 1A(7) (lines 18–19). At this point, all processes trust p_1, all have bal $= 1$, and the only messages in transit are 1A(4) and 1A(7) from p_1. We refer to this point in the execution as ($*$).

Deliver 1A(4) and 1A(7) to p_1 in order. It accepts both, updating bal $= 7$ (line 13). Now assume that p_2 and p_3 receive 1A(7) before 1A(4). Since p_1 is trusted and $7 >$ bal $= 1$, they accept it and update bal $= 7$ (line 13). Later, when they receive 1A(4), they reject it and send a NACK to p_1 (line 15).

Deliver the NACKs to p_1. In response, it satisfies the guard at line 17 twice, increments next_bal to 10 and then to 13 (line 18), and broadcasts 1A(10) and 1A(13) (line 19). At this point, all processes trust p_1, all have bal $= 7$, and the only messages in transit are 1A(10) and 1A(13) from p_1. This matches the previous configuration ($*$), and the cycle can repeat indefinitely.

This violates Eventual Leadership, as the system never converges on a final ballot. The root cause is that every NACK – whether caused by a delayed or outdated ballot – is treated as evidence of failure, triggering unnecessary escalation.

```
1  bal $\leftarrow 1$
2  next_bal $\leftarrow i$
3  trusted $\leftarrow p_1$

4  on trust($p$)
5      if $p \neq$ trusted then
6          send NACK($\infty$) to trusted
7      trusted $\leftarrow p$
8      if $p = p_i$ then
9          next_bal $\leftarrow$ next_bal $+ |\mathcal{P}|$
10         send 1A(next_bal) to all

11 when received 1A($b$) from $p_j$
12     if trusted $\neq p_j$ then
13         send NACK($\infty$) to $p_j$
14     else if $b <$ bal then
15         send NACK(bal) to $p_j$
16     else
17         bal $\leftarrow b$

18 when received NACK($b$)
19     if trusted $= p_i \land b >$ next_bal then
20         next_bal $\leftarrow$ next_bal $+ |\mathcal{P}|$
21         send 1A(next_bal) to all
```

Figure 2.9: Ballot change mechanism with our fix for suppressing stale NACKs.

Refined Handling of NACK Messages. To prevent ballot growth due to obsolete NACKs, we refine the protocol to distinguish between valid and stale rejections. Specifically, each NACK now includes the sender's current ballot value bal (line 15) or ∞ (lines 6 and 13). Upon receiving a NACK(b), a trusted leader process p_i reacts only if the received ballot b exceeds its own next_bal (line 19). This guard ensures that the rejection indeed indicates that the leader must increase its ballot to reach all correct processes. If the condition fails, it means the leader has already sent a 1A message with a ballot number higher than the one reported, and no update is needed.

This change prevents feedback loops caused by delayed 1A messages. Consider again the previous counterexample: 1A(4) is delivered after 1A(7) has already been accepted, triggering stale NACKs that unnecessarily cause p_1 to increase its ballot again. With the new guard at line 19, such rejections no longer lead to ballot escalation, as they correspond to ballot values that do not exceed the leader's current next_bal. In this example, the second rejection received by p_1 is NACK(7), but p_1 has already updated next_bal $= 7$ after processing the first rejection. Since $7 \not> 7$, the guard fails and no new ballot is initiated.

Alternatively, the issue could be avoided by assuming FIFO channels, which would ensure that 1A messages are received in order and cannot cause rejections based on outdated ballots. However, our solution achieves the same effect without imposing additional assumptions.

Correctness. We now show that the protocol in Figure 2.9 satisfies Eventual Leadership. Let q denote the correct process that is eventually trusted by all correct processes, as guaranteed by Ω. We prove that all correct processes eventually adopt the same ballot led by q.

Lemma 2.3. *There exists a time after which only q can increase its ballot* next_bal.

Proof. Let t be a time after which all correct processes permanently trust q. From that time on, the guards in line 8 (local trust) and line 19 ($\text{trusted} = p_i$) can be satisfied only at q. Therefore, only q can increase its ballot. □

Lemma 2.4. *Process q broadcasts only finitely many* `1A` *messages.*

Proof. Process q broadcasts `1A` only in the following cases:

1. At line 8, when q becomes trusted. Since Ω converges, this can occur only finitely many times.
2. At line 19, upon receiving a `NACK`(b) with $b >$ next_bal. Such `NACK`s are of two kinds:
   - Those from lines 6 and 13, which require the sender not to trust q. These vanish once Ω converges, so only finitely many exist.
   - Those from line 15, meaning the sender had $\text{bal} = b$ when sending `NACK`(b). Since $b >$ next_bal at q and ballots created by q are never larger than its current next_bal, ballot b must have been created by some process other than q. By Lemma 2.3, eventually no process other than q can create new ballots. Hence only finitely many such ballots exist, and q can increase past each at most once.

Therefore q broadcasts `1A` only finitely many times, as required. □

Theorem 2.5. *There exists a ballot $b^\star$ led by q such that eventually every correct process adopts $b^\star$ and never abandons it.*

Proof. Let $b^\star$ be the maximum ballot ever broadcast by q in a `1A` message, guaranteed to exist by Lemma 2.4. When sending this message, q must trust itself, as ensured by the guards at lines 8 and 19. Moreover, q cannot later stop trusting itself. If it did, then – since q is eventually trusted permanently by itself – it would eventually retrust itself. This would cause a new `1A` with ballot greater than $b^\star$ (line 10), contradicting maximality.

Now consider an arbitrary correct process p that receives `1A`$(b^\star)$. If p does not trust q, or if $b^\star <$ bal at p, then p replies with a `NACK`(b') for some $b' > b^\star$. Since next_bal is the last value sent in an `1A`, at this point q has $\text{next\_bal} = b^\star$ and still trusts itself. Thus, receiving such a `NACK`(b') would satisfy line 19 at q and trigger a new `1A` with ballot greater than $b^\star$ – again a contradiction. Hence p must adopt $b^\star$ (line 17).

Suppose now that some correct process other than q that has adopted $b^\star$ later stops trusting q. Then line 6 would send a `NACK`(∞) to q, which would again trigger line 19 and force a ballot greater than $b^\star$ – contradicting maximality. Hence any process that adopts $b^\star$ continues to trust q forever. Moreover, no process can adopt another ballot: doing so requires trusting its sender, but every correct process trusts q, and q never broadcasts a different ballot. □

```
function propose($x$):
    my_val $\leftarrow x$
    async wait until phase $=$ DECIDE
    return val
when received 1A($b$) from $p_j$
    if trusted $\neq p_j$ then
        send NACK($\infty$) to $p_j$
    else if $b <$ bal then
        send NACK(bal) to $p_j$
    else
        bal $\leftarrow b$
        phase $\leftarrow$ ELECTION
        send 1B($b$, cbal, val) to $p_j$
```

Figure 2.10: Modification of the handlers of propose(x) and receipt of 1A(b).

Putting it together. Finally, to obtain the desired live implementation of consensus, we merge the code in Figures 2.5 and 2.9, modifying the handlers for propose(x) and for the reception of 1A(b) as shown in Figure 2.10.

Chapter 3

Modeling Framework and Core Contributions

This chapter introduces the computational framework used throughout the thesis and presents our core contributions. We begin by defining a system model that accounts for adversarial channel behavior. We then extend the notion of fail-prone systems to include channel failures and introduce a novel liveness condition tailored to this setting. Building on these foundations, we introduce the abstraction of generalized quorum systems (GQS), which characterize the reliability required to solve classical problems under arbitrary fail-prone systems. Finally, we present our main results, showing that a GQS is necessary and sufficient to implement atomic registers, atomic snapshots, lattice agreement, and consensus in our adversarial setting.

3.1 System Model

We consider a message-passing system consisting of a set of n processes $\mathcal{P}$ 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.

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.

Our lower bounds assume the stronger notion of channel correctness (reliable) and the more restrictive notion of faultiness (disconnected), whereas our upper bounds assume the weaker notion of channel correctness (eventually reliable) and the broader notion of 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 (see Figure 1.3).

We use the *asynchronous* model in our results about registers, snapshots and lattice agreement, and the *partially synchronous* model [24, 32] in our results about consensus. The latter 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 unknown.

3.2 Fail-Prone Systems under Channel Failures

To state our results, we need a way of specifying which processes and channels may fail during an execution. We do this by generalizing the classical notion of a fail-prone system [52] (see §2.4) to include channel failures. Our formulation is generic, irrespective of a specific process or channel failure type. The exact failure types are defined when we instantiate the system model to present our results.

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. Here, C contains only channels between correct processes, as channels incident to faulty processes are considered faulty by default:

$$(p, q) \in C \implies \{p, q\} \cap P = \emptyset.$$

Given a failure pattern $f = (P, C)$, an execution σ is *f-compliant* if at most the processes in P and the channels in C fail in σ. A *fail-prone system* $\mathcal{F}$ is then defined as a set of such failure patterns (see Figure 1.2). In this way, the extended notion captures a broader range of failure scenarios, including arbitrary combinations of process and channel failures.

Example 3.1. The standard failure model where any minority of processes can fail and channels between correct processes are correct is captured by the fail-prone system

$$\mathcal{F}_M = \{(Q, \emptyset) \mid Q \subseteq \mathcal{P} \wedge |Q| \leq \lfloor \tfrac{n-1}{2} \rfloor\}.$$

3.3 Liveness under Channel Failures

Defining liveness properties under our failure model is subtle, since channel failures may isolate some correct processes, making it impossible to ensure termination at all of them. To address this, we adapt the classical notions of obstruction-freedom and wait-freedom (§2.3) by parameterizing them based on the allowed failures and the corresponding 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 fair 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 1.2 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 [13, 34, 41].

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

Example 3.2. The standard guarantee of wait-freedom under a minority of process failures corresponds to $(\mathcal{F}_M, \tau_M)$-wait-freedom, where $\mathcal{F}_M$ is defined in Example 3.1 and τ_M selects the set of all correct processes:

$$\forall f = (P, \emptyset) \in \mathcal{F}_M.\, \tau_M(f) = \mathcal{P} \setminus P.$$

3.4 Generalized Quorum Systems

The existence of a classical quorum system (§2.5) is sufficient to implement atomic registers, atomic snapshots, lattice agreement and consensus in a model without channel failures. We now generalize this notion to account for adversarial channel behavior.

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 or Paxos. Surprisingly, we find that this connectivity requirement is unnecessarily restrictive: the above-listed problems can be solved even when read quorums are not strongly connected.

To define the extended notion of a quorum system suitable for our failure model, we introduce the following concepts.

- *Network graph:* let $\mathcal{G} = (\mathcal{P}, \mathcal{C})$ be the directed graph with processes as vertices and 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.

- *f-availability:* a set Q is f-available if it consists of correct processes and 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 is f-reachable from a set R if both W and R are 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 3.3. In Figure 1.2, for each $i = 1..4$, the write quorum W_i is f_i-available and f_i-reachable from the read quorum R_i.

Definition 3.4. *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, Availability ensures that under any failure scenario described by $\mathcal{F}$, an operational write quorum can unidirectionally receive information from an operational read quorum. A classical quorum system is a special case of a generalized quorum system. Indeed, if $\mathcal{F}$ disallows channel failures between correct processes, then any correct write quorum can trivially be reached from any correct read quorum, and Definition 3.4 reduces to Definition 2.1.

Example 3.5. Consider the fail-prone system $\mathcal{F} = \{f_i \mid i = 1..4\}$ in Figure 1.2 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.

The above $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ is a valid generalized quorum system even though processes in each read quorum are not strongly connected via correct channels. This relaxation allows read quorums to form 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 all write quorums validating Availability with respect to f must also be strongly connected via correct channels.

Proposition 3.6. *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 $\mathcal{G} \setminus f$:*

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

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

- $(p_2 \in W_2) \wedge (W_2$ is f-available$) \wedge \exists R_2 \in \mathcal{R}. (W_2$ is f-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. □

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 $U_f \neq \emptyset$.

3.5 Main Results

We now formally state our main results, which are proved in the remainder of the thesis. We show that the existence of a generalized quorum system is a tight bound on the combinations of process and channel failures that can be tolerated by any implementation of MWMR atomic registers, SWMR atomic snapshots, and lattice agreement.

The following theorem, proved in §4.1, establishes that the existence of a generalized quorum system is necessary to implement any of these objects under the model $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected). This holds under a weak termination guarantee that only requires obstruction-freedom at some non-empty set of processes for each failure pattern.

Theorem 3.7. *Let $\mathcal{F}$ be a fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be a termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) \neq \emptyset$. Assume that there exists an $(\mathcal{F}, \tau)$-obstruction-free implementation of any of the following objects over the model $\mathcal{M}_{\text{ARD}}$: MWMR atomic registers, SWMR 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 next theorem, proved in §4.3, establishes a matching upper bound under the more adversarial model $\mathcal{M}_{\text{AEF}}$ (asynchronous / eventually reliable / flaky). It shows 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 U_f.

Theorem 3.8. *Let $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ be a generalized quorum system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be the termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) = U_f$. Then there exists an $(\mathcal{F}, \tau)$-wait-free implementation for each of the following objects over the model $\mathcal{M}_{\text{AEF}}$: MWMR atomic registers, SWMR atomic snapshots, and lattice agreement.*

Theorem 3.7 shows that U_f is the largest set of processes for which termination can be guaranteed under the failure pattern f. Thus, the two theorems together imply that if termination can be ensured for at least one correct process under f, then it can also be ensured for all processes in U_f.

Example 3.9. Consider the generalized quorum system $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ from Figure 1.2. 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 3.8, there exists an $(\mathcal{F}, \tau)$-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 3.7 establishes that no implementation of any of the three objects can guarantee obstruction-freedom, and by extension wait-freedom, at any process under $\mathcal{F}'$.

Our results consider the weakest variant of lattice agreement [14] (see §2.6.4), which is single-shot and therefore cannot be used for implementing multi-shot objects, such as registers. Thus, to prove the lower bound (Theorem 3.7), we first establish it for lattice agreement. Since SWMR atomic snapshots can be constructed from MWMR atomic registers (§2.6.3), and lattice agreement can in turn be constructed from snapshots (§2.6.4), we naturally get the lower bounds for the former two problems. As a byproduct of these, we derive in §4.2 a more precise formulation of the celebrated CAP theorem.

To prove the upper bound (Theorem 3.8), we construct an $(\mathcal{F}, \tau)$-wait-free implementation of MWMR atomic registers; then the above-mentioned constructions imply the upper bounds for snapshots and lattice agreement.

Next, we show that the existence of a generalized quorum system also is a tight bound on the combinations of process and channel failures that can be tolerated by any implementation of consensus in the partially synchronous model.

The following theorem, proved in §5.1, establishes that the existence of a generalized quorum system is necessary to implement consensus under the model $\mathcal{M}_{\text{PRD}}$ (partially synchronous / reliable / disconnected). Like Theorem 3.7, it only requires obstruction-freedom to hold at some non-empty set of processes for each failure pattern.

Theorem 3.10. *Let $\mathcal{F}$ be a fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be a termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) \neq \emptyset$. If there exists an $(\mathcal{F}, \tau)$-obstruction-free implementation of consensus over the model $\mathcal{M}_{\text{PRD}}$, 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 $\tau(f) \subseteq U_f$.*

Finally, the next theorem, proved in §5.2, establishes a matching upper bound under the more adversarial model $\mathcal{M}_{\text{PEF}}$ (partially synchronous / eventually reliable / flaky). It shows that the existence of a generalized quorum system is sufficient to implement consensus. Like Theorem 3.8, for each $f \in \mathcal{F}$, the implementation guarantees wait-freedom within U_f.

Theorem 3.11. *Let $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ be a generalized quorum system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be the termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) = U_f$. Then there exists an $(\mathcal{F}, \tau)$-wait-free implementation of consensus over the model $\mathcal{M}_{\text{PEF}}$.*

The upper bounds in Theorems 3.8 and 3.11 establish the tightness of our lower bounds, settling the computability question. However, the associated algorithms are inefficient. To improve efficiency, we consider a practically motivated class of systems where at most a minority of processes may fail. This restriction enables stronger connectivity guarantees (§4.4.1, §5.3) and supports more efficient implementations with bounded operation latency and memory usage (§4.4.2, §5.4).

3.6 Comparison with Existing Results

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 [4, 27, 28, 58, 62, 63, 64]. The seminal paper by Dolev [29] and subsequent work (see [61] for a 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 bi-directionally connected.

Early models, such as *send omission* [39] and *generalized omission* [59], extended crash failures by allowing processes to fail in sending or receiving messages. These models were shown to be computationally equivalent to crash failures in both synchronous [56] and asynchronous [26] 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 [62], 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 [62, 63]. 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 in [7, 30, 31, 37] provided a majority of correct processes is strongly connected and can eventually communicate in a timely fashion. In contrast, we show that much weaker connectivity constraints, captured via GQS, are necessary and sufficient for implementing both register and consensus. Aguilera et al. [6] and subsequent work [5, 33, 43, 51] studied the implementation of Ω (§2.7.2) – the weakest for consensus [23] – 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, in §5.5 we will prove them to be insufficient for consensus.

Alquraan et al. [10] presented a study of system failures due to faulty channels, which we already mentioned in Chapter 1. This work highlights the practical importance of designing provably correct systems that explicitly account for channel failures. Follow-up work [9, 57] proposed practical systems for tolerating channel failures, but did not investigate optimal fault-tolerance assumptions.

Chapter 4

Atomic Registers, Atomic Snapshots and Lattice Agreement

In this chapter, we study the implementation of multi-writer multi-reader atomic registers, single-writer multi-reader atomic snapshots, and lattice agreement in asynchronous systems subject to arbitrary process and channel failures. We show that a generalized quorum system (GQS), as introduced in Chapter 3, provides a tight reliability bound for solvability: such an object can be implemented if and only if the underlying fail-prone system admits a GQS.

The main complexity lies in establishing the upper bounds (see Example 1.3). Classical implementations of these objects for models with reliable connectivity, such as those presented in §2.6, rely on bidirectional connectivity between quorums – for example, by following a traditional request/response pattern of quorum access. In contrast, our failure model allows processes to have only unidirectional connectivity or none at all, invalidating these assumptions and making classical algorithms inapplicable.

This asymmetry poses a significant challenge for implementing these objects, as it rules out symmetric communication patterns. To address this, in §4.3 we introduce novel logical clocks that enable write and read quorums to reliably track state updates without requiring bidirectional communication. These clocks provide the missing coordination mechanism for implementing shared objects in settings previously considered too unreliable.

Having established tight bounds for solvability, in §4.4 we turn our attention to a practically motivated class of fail-prone systems in which at most a minority of processes may fail. While this assumption narrows the set of admissible failure patterns, it yields stronger connectivity guarantees that we leverage to design more efficient implementations – ones where operation latency can be bounded by message delays and state exchange rates. These implementations also use bounded memory, reflecting practical system constraints.

Finally, as a byproduct of the lower bounds obtained in §4.1, we derive in §4.2 a more precise formulation of the celebrated CAP theorem. While the classical version states that wait-free atomic registers cannot be implemented under arbitrary message loss, the resulting bound is not tight. Our result refines this by identifying exactly where availability can be guaranteed: termination is possible only at processes that are strongly connected by correct channels.

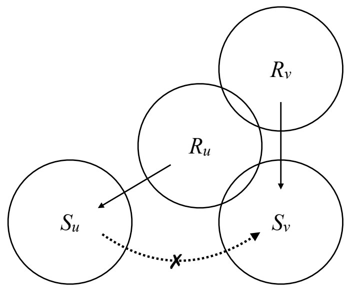

Figure 4.1: Illustration of the sets R_k and S_k for $k \in \{u, v\}$.

4.1 Lower Bound for Lattice Agreement

We now prove Theorem 3.7 for lattice agreement. Then the existing constructions of snapshots from registers (§2.6.3) and of lattice agreement from snapshots (§2.6.4) imply that the theorem also holds for these objects. Our proof relies on the following lemma, which establishes that processes where obstruction-freedom holds must be strongly connected by correct channels.

Lemma 4.1. *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 over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), then T is strongly connected in $\mathcal{G} \setminus f$.*

Proof. Assume by contradiction that $\mathcal{A}$ is an (f, T)-obstruction-free implementation of lattice agreement, but T is not strongly connected in $\mathcal{G} \setminus f$. Thus, there exist $u, v \in T$ such that there is no path from u to v in $\mathcal{G} \setminus f$, or from v to u. Without loss of generality, assume the former. Let S_u be the strongly connected component of $\mathcal{G} \setminus f$ containing u, and S_v be the strongly connected component of $\mathcal{G} \setminus f$ containing v. By assumption, $S_u \cap S_v = \emptyset$. Let R_u be the set of processes outside S_u that can reach u in $\mathcal{G} \setminus f$, and R_v be the set of processes outside S_v that can reach v in $\mathcal{G} \setminus f$. Refer to Figure 4.1 for a visual depiction.

The next claims follow directly from the definitions of R_k and S_k for each $k \in \{u, v\}$.

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 $\mathcal{G} \setminus f$.*

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

Proof of Claim 3. Assume by contradiction that $S_u \cap (R_v \cup S_v) \neq \emptyset$. Let $w \in S_u \cap (R_v \cup S_v)$. Since $w \in S_u$, there exists a path from u to w. Since $w \in R_v \cup S_v$, there exists a path from w to v. Concatenating these paths creates a path from u to v, contradicting the assumption that no such path exists. □

Let $\mathcal{L}$ be a semi-lattice with partial order $\leq$ such that $x_u, x_v \in \mathcal{L}$, $x_u \not\leq x_v$ and $x_v \not\leq x_u$. Let α_1 be a fair execution of $\mathcal{A}$ where the processes and channels in f fail at the beginning, process u invokes $\mathsf{propose}(x_u)$, and no other operation is invoked in α_1. Because $u \in T$ and $\mathcal{A}$ is (f, T)-obstruction-free, the $\mathsf{propose}(x_u)$ operation must eventually terminate with a return value y_u. By Downward validity (see §2.6.4), $x_u \leq y_u$. By Upward validity, since no other propose() was invoked, $y_u \leq x_u$. Thus, $y_u = x_u$. Let α_2 be the prefix of α_1 ending with the response to propose(). By Claim 1, $R_u \cup S_u$ is unreachable from $\mathcal{P} \setminus (R_u \cup S_u)$, and 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 $\alpha_3 = \alpha_2|_{R_u \cup S_u}$, the projection of α_2 to actions by processes in $R_u \cup S_u$, is an execution of $\mathcal{A}$. By Claim 2, R_u is unreachable from S_u, and 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 an execution of $\mathcal{A}$. Finally, $\alpha_3|_{R_u}$ contains no propose invocations (only startup steps).

Let β_1 be a fair execution of $\mathcal{A}$ that starts with all the actions from $\alpha_3|_{R_u}$, followed by the failure of all processes and channels in f, followed by a propose(x_v) invocation by the process v. Because $v \in T$ and $\mathcal{A}$ is (f, T)-obstruction-free, the propose(x_v) operation must eventually terminate with a return value y_v. By Downward validity, $x_v \leq y_v$. By Upward validity, since no other propose() was invoked, $y_v \leq x_v$. Thus, $y_v = x_v$. Let β_2 be the prefix of β_1 ending with the response to propose(). 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 in β_2 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 no process or channel fails. 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

$$\sigma|_{R_u \cup R_v \cup S_v} = (\alpha_3|_{R_u}\delta\alpha_3|_{S_u})|_{R_u \cup R_v \cup S_v} = \alpha_3|_{R_u}\delta = \beta.$$

Thus, σ is indistinguishable from β to the processes in $R_u \cup R_v \cup S_v$. Also,

$$\sigma|_{S_u} = (\alpha_3|_{R_u}\delta\alpha_3|_{S_u})|_{S_u} = \alpha_3|_{S_u} = (\alpha_3|_{R_u}\alpha_3|_{S_u})|_{S_u} = \alpha|_{S_u}.$$

Thus, σ is indistinguishable from α to the processes in S_u. Finally, $\sigma|_{\mathcal{P} \setminus (R_u \cup S_u \cup R_v \cup S_v)} = \epsilon$. 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 u decides x_u, v decides x_v, and x_u, x_v are incomparable in $\mathcal{L}$. This contradicts the Comparability property of lattice agreement. The contradiction derives from assuming that T is not strongly connected in $\mathcal{G} \setminus f$, so the required follows. □

Proof of Theorem 3.7 for lattice agreement. Let $\mathcal{A}$ be an $(\mathcal{F}, \tau)$-obstruction-free implementation of lattice agreement. For a failure pattern $f \in \mathcal{F}$, Lemma 4.1 implies that the set $\tau(f)$ of processes where obstruction-freedom 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

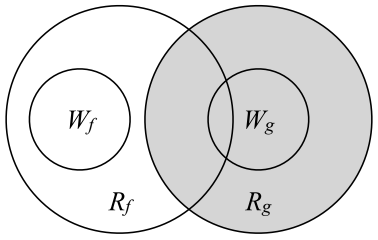

Figure 4.2: Illustration of the sets W_k and R_k for $k \in \{f, g\}$.

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 $W_f \cap R_g = \emptyset$. Refer to Figure 4.2 for a visual depiction.

The next claims follow directly from the definitions of R_k and W_k, for each $k \in \{f, g\}$.

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

Claim 2. *For any $k \in \{f, g\}$, $R_k \setminus W_k$ 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 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 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 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 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 $\alpha_3 = \alpha_2|_{R_f}$ 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 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 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 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 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:

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

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

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

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 $\tau(f) \subseteq W_f \subseteq U_f$. $\square$

4.2 CAP Revisited

Before turning our attention to the upper bounds, we first highlight a few corollaries of our lower bounds. These results are of independent interest because they generalize the celebrated CAP theorem [19], which asserts that no system can simultaneously guarantee consistency, availability, and network partition tolerance.

The CAP formalization by Gilbert and Lynch [38] 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 may lose any subset of messages (in our terminology, flaky).

Theorem 4.2 (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 merely states that, to implement a wait-free atomic register, *some* channels must be correct, but clearly a single correct channel is not sufficient in general. Second, CAP requires availability at *all* processes, and is thus not applicable when aiming to ensure availability in only a *part* of the system, as formalized by the liveness notions in §3.3.

Our first result lifts these limitations and further strengthens CAP by considering a stronger model, $\mathcal{M}_{\mathrm{ARD}}$ (asynchronous / reliable / disconnected), where channels may only fail by disconnection. Informally, we show that processes where obstruction-freedom holds must be transitively connected via correct channels – no such process can be partitioned from the rest.

Corollary 4.3. *Let f be a failure pattern and $T \subseteq \mathcal{P}$. If some algorithm $\mathcal{A}$ is an (f, T)-obstruction-free implementation of an atomic register over $\mathcal{M}_{\mathrm{ARD}}$, then T is strongly connected in $\mathcal{G} \setminus f$.*

The corollary follows from Lemma 4.1 via reduction: the constructions of snapshots from registers (§2.6.3) and of lattice agreement from snapshots (§2.6.4) imply that registers can be used to solve lattice agreement. Thus, if the lower bound holds for lattice agreement, it must also hold for registers.

In turn, the above 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 4.4. *Let f be a failure pattern that disallows process failures: $\exists C \subseteq \mathcal{C}.\, f = (\emptyset, C)$.*

If some algorithm $\mathcal{A}$ is an $(f, \mathcal{P})$-obstruction-free implementation of an atomic register over $\mathcal{M}_{\mathrm{ARD}}$, then the graph $\mathcal{G} \setminus f$ is strongly connected.

This implies CAP (Theorem 4.2): if an algorithm $\mathcal{A}$ implements a wait-free atomic register, then it also implements an obstruction-free atomic register, and by the above corollary, all processes must be strongly connected by correct channels.

Corollary 4.4 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 4.5. *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 $\mathcal{A}$ is an $(\mathcal{F}, \tau_{\mathcal{P}})$-obstruction-free implementation of an atomic register over $\mathcal{M}_{\mathrm{ARD}}$, 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 $\mathcal{M}_{\mathrm{AEF}}$ (asynchronous / eventually reliable / flaky). The proposition follows directly from our more general upper bound (Theorem 3.8).

Proposition 4.6. *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 $(\mathcal{F}, \tau_{\mathcal{P}})$-wait-free implementation of an atomic register over $\mathcal{M}_{\mathrm{AEF}}$.*

4.3 Upper Bound for Atomic Registers

Fix a generalized quorum system $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ and a termination mapping $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ such that $\tau(f) = U_f$ for each $f \in \mathcal{F}$, where U_f is defined after Proposition 3.6. Without loss of generality, we assume that for each $f \in \mathcal{F}$, the connectivity relation of the graph $\mathcal{G} \setminus f$ is transitive; if not, transitivity can be easily simulated by having all processes forward every received message.

To implement atomic registers using the generalized quorum system, each process in the termination set must be able to retrieve information from both a read and a write quorum – a common pattern in algorithms like ABD [12]. This is challenging in our setting, since processes within a read quorum may not be strongly connected by correct channels. As a result, some members may lack incoming correct channels from other processes and be unable to receive their messages.

Example 4.7. Consider the generalized quorum system in Figure 1.2 and the failure pattern f_1. The available read and write quorums are $R_1 = \{a, c\}$ and $W_1 = \{a, b\}$, respectively. Since Theorem 3.8 requires $\tau(f_1) = U_{f_1}$, any register implementation that satisfies the theorem must ensure wait-freedom within W_1 – in particular, at a. However, all channels incoming to process c may have failed. This makes it impossible for some members of W_1, such as a, to request information from some members of R_1, such as c, by sending them an explicit message to this end.

4.3.1 Quorum Access Functions

We encapsulate the mechanisms needed to address these challenges (Examples 1.3 and 4.7) 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 local state from a set $\mathcal{S}$ at each process. The interface of the quorum access functions is as follows:

- quorum_get() $\in 2^{\mathcal{S}}$: returns the states of all members of some read quorum; and
- quorum_set(u): applies the *update function* $u : \mathcal{S} \to \mathcal{S}$ to the states of all members of some write quorum. We assume that update functions are uniquely identifiable.

We require that these functions satisfy the following properties:

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

Note that the uniqueness of update functions guarantees that the specification is well-defined. As we show in the following, quorum access functions are sufficient to construct an ABD-like algorithm for atomic registers.

Quorum access functions for classical quorum systems. To illustrate the concept of quorum access functions, in Figure 4.3 we provide their implementation under a classical quorum system that disallows channel failures (Definition 2.1).

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: 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 quorum_get() at a process, it broadcasts a GET_REQ message (line 5). Any process receiving this request responds with a GET_RESP message that carries its current state (line 9). The quorum_get() invocation returns once it accumulates such responses from a read quorum (line 6). It is precisely here that this implementation requires a process at which termination must be ensured to have bidirectional reliable communication with every member of some read quorum, implying that the read quorum must be strongly connected by reliable channels. Also, to cope with message loss on channels that are only eventually reliable, the messages GET_REQ are periodically retransmitted (line 5).

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

Figure 4.3: Quorum access functions for a classical quorum system: the protocol at a process p_i.

Upon a call to quorum_set(u) at a process, it broadcasts a SET_REQ message carrying the update function u (line 12). Any process receiving this request applies u to its current state and responds with SET_RESP (line 16). The quorum_set() invocation returns once it accumulates such responses from a write quorum (line 13). Like before, to cope with message loss, the messages SET_REQ are periodically retransmitted (line 12).

It is easy to see that the above implementation satisfies both Validity and Real-time ordering. The latter follows from the fact that any read and write quorums intersect. Since a call to quorum_set(u) completes only after applying the update function u to all members of some write quorum, any subsequent call to quorum_get() will collect responses from a read quorum that intersects the earlier write quorum. Therefore, at least one member of the read quorum (i.e., one in the intersection) must reflect u and return it in its response. Validity holds because each quorum_get() collects states from a read quorum, and every such state results from applying a subset of past update functions in some order.

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 correct process can communicate with both of 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 4.7 for illustration. Recall that in this example we need to ensure termination

```
seq ← 0
clock ← 0
state ∈ $\mathcal{S}$   // *opaque state of the top-level protocol*
**function** quorum_get():
    seq ← seq + 1
    **periodically send** CLOCK_REQ(seq) **to all**
    **wait until received** $\{$CLOCK_RESP(seq, $c_j$) $\mid p_j \in W_{\text{get}}\}$ **from some** $W_{\text{get}} \in \mathcal{W}$
    $c_{\text{get}} \leftarrow \max\{c_j \mid p_j \in W_{\text{get}}\}$
    **wait until received** $\{$GET_RESP($s_j, c_j$) $\mid p_j \in R_{\text{get}}\}$ **from some** $R_{\text{get}} \in \mathcal{R}$
        **where** $\forall j.\ c_j \geq c_{\text{get}}$
    **return** $\{s_j \mid p_j \in R_{\text{get}}\}$
**when received** CLOCK_REQ($k$) **from** $p_j$
    **send** CLOCK_RESP($k$, clock) **to** $p_j$
**periodically**
    clock ← clock + 1
    **send** GET_RESP(state, clock) **to all**
**function** quorum_set($u$):
    seq ← seq + 1
    **periodically send** SET_REQ(seq, $u$) **to all**
    **wait until received** $\{$SET_RESP(seq, $c_j$) $\mid p_j \in W_{\text{set}}\}$ **from some** $W_{\text{set}} \in \mathcal{W}$
    $c_{\text{set}} \leftarrow \max\{c_j \mid p_j \in W_{\text{set}}\}$
    **wait until received** $\{$GET_RESP(_, $c_j$) $\mid p_j \in R_{\text{set}}\}$ **from some** $R_{\text{set}} \in \mathcal{R}$
        **where** $\forall j.\ c_j \geq c_{\text{set}}$
**when received** SET_REQ($k, u$) **from** $p_j$
    state ← $u$(state)
    clock ← clock + 1
    **send** SET_RESP($k$, clock) **to** $p_j$
```

Figure 4.4: Quorum access functions for a generalized quorum system: protocol at a process p_i.

within W_1 and, in particular, at a. To achieve this, 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 invoked 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, which is different from the usual logical clocks for tracking causality [35, 45], is used by processes to tag their state. We then modify the implementation of the quorum access functions in Figure 4.3 as shown in Figure 4.4:

- **Periodic state propagation (line 13):** A process periodically advances its clock and propagates its current state along with clock in a GET_RESP message, without waiting for an explicit request. This is how the clock value tags the state and indicates the logical time at which the process held it.
- **Clock updates during state changes (line 22):** 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 16):** 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 observe the update u.
- **Clock cutoff for quorum_get (line 4):** 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 states s_j to the caller.

As in the implementation for classical quorum systems, to tolerate message loss on eventually reliable channels, processes periodically retransmit the CLOCK_REQ and SET_REQ messages (lines 6 and 18).

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 17-20). The following lemma shows that querying the states of a read quorum with clocks $\geq c_{\text{set}}$ is sufficient to observe u.

Lemma 4.8. *Assume that c_{set} is computed by a* quorum_set(u) *invocation at a process p_{set} (lines 17-20). Consider a set of messages* $\{$GET_RESP$(s_j, c_j) \mid p_j \in R\}$ *sent by all members of* *some read quorum R, each having $c_j \geq c_{\text{set}}$. 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 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 24 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. □

In the light of the above lemma, for 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 quorum_set(). As we show in the following proof, this is precisely what is achieved by the computation of c_{get} in quorum_get().

Theorem 4.9 (Real-time ordering)**.** *If a* quorum_set(u) *operation terminates, then its effect is visible to any subsequent* quorum_get() *invocation.*

Proof. Assume that quorum_set(u) terminates at a process p_{set} before 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 quorum_set(u) terminated, p_{set} received `GET_RESP`$(\_, c'_l)$ from p_l with $c'_l \geq c_{\text{set}}$ (line 21). Hence, by the time 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 7. This message was sent at line 12 after quorum_get() had been invoked, and thus, after 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 8, 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 9 satisfies $c_j \geq c_{\text{get}} \geq c_{\text{set}}$. By Lemma 4.8, some s_j has incorporated u, as required. □

Theorem 4.10 (Liveness)**.** *The protocol in Figure 4.4 is $(\mathcal{F}, \tau)$-wait-free.*

Proof. We prove the liveness of quorum_get(); the case of quorum_set() is analogous. Fix a failure pattern $f \in \mathcal{F}$ and a process $p \in \tau(f)$ that executes 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 3.6, $W \subseteq U_f$. 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 eventually exchange the `CLOCK_REQ` and `CLOCK_RESP` messages with every member of W, thus exiting the wait at line 7. 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 13). Hence, p will eventually receive `GET_RESP` messages from all members of R with high enough clock values to exit the wait at line 9 and return. □

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

Figure 4.5: Atomic Register protocol at a process p_i.

4.3.2 Registers via Quorum Access Functions

In Figure 4.5 we give an implementation of an atomic register using the above quorum access functions, which validates Theorem 3.8. The implementation follows the structure of a multi-writer multi-reader variant of ABD [12, 50] (see §2.6.2): the main novelty of our protocol lies in the implementation of the quorum access functions.

Let Value be the domain of values the register stores. Like ABD, our implementation tags values with versions from Version $= \mathbb{N} \times \mathbb{N}$, which are 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 $(\mathsf{val}, \mathsf{ver})$, 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 4.4 to $\mathcal{S} = \mathsf{Value} \times \mathsf{Version}$, with $(\bot, 0)$ as the initial state. Note that versions are unrelated to the logical clocks in the implementation of the quorum access functions.

To execute a write(x) operation (line 2), a process proceeds in two phases:

- **Get phase.** The process uses 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 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 $s.\mathsf{ver}$, 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 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 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'.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.

4.3.3 Correctness

The Liveness property of the quorum access functions trivially implies that the protocol in Figure 4.5 is $(\mathcal{F}, \tau)$-wait-free. We now show that the protocol in Figure 4.5 is linearizable [40], thereby completing the proof of Theorem 3.8.

To each (possibly infinite) execution σ of the algorithm, we associate:

- a set $V(\sigma)$ consisting of all the operations in σ, i.e., reads and writes; and
- a relation $\mathsf{rt}(\sigma) \subseteq V(\sigma) \times V(\sigma)$, defined as follows: for all $o_1, o_2 \in V(\sigma)$, $(o_1, o_2) \in \mathsf{rt}(\sigma)$ if and only if o_1 completes before o_2 is invoked.

We denote the read operations in σ by $R(\sigma)$ and the write operations in σ by $W(\sigma)$.

The following definition and theorem are inspired by the dependency graph framework introduced by Adya [1].

Definition 4.11. *Let σ be an execution. A **dependency graph** for σ is a tuple $G = (V(\sigma), \mathsf{vis}, \mathsf{rt}(\sigma), \mathsf{wr}, \mathsf{ww}, \mathsf{rw})$, where vis is a **visibility** predicate over $V(\sigma)$ and the relations $\mathsf{wr}, \mathsf{ww}, \mathsf{rw} \subseteq V(\sigma) \times V(\sigma)$ are such that:*

1. *vis holds for all complete operations;*
2. *(i) if $(w, r) \in \mathsf{wr}$, then $w \in W(\sigma)$, $\mathsf{vis}(w)$, $r \in R(\sigma)$, r completes and $\mathsf{val}(w) = \mathsf{val}(r)$;*

   (ii) for all $w_1, w_2, r \in V(\sigma)$ such that $(w_1, r) \in \mathsf{wr}$ and $(w_2, r) \in \mathsf{wr}$, we have $w_1 = w_2$;

   (iii) if $r \in R(\sigma)$ is complete and there is no $w \in W(\sigma)$ such that $(w, r) \in \mathsf{wr}$, then r returns the initial value $\bot$; and

   (iv) if $w \in W(\sigma)$ is incomplete and there is no $r \in R(\sigma)$ such that $(w, r) \in \mathsf{wr}$, then $\neg\mathsf{vis}(w)$;
3. *ww is a total order over $\{w \mid w \in W(\sigma) \land \mathsf{vis}(w)\}$; and*
4. *$\mathsf{rw} = \{(r, w) \mid \exists w'.\ (w', r) \in \mathsf{wr} \land (w', w) \in \mathsf{ww}\} \cup$
   $\{(r, w) \mid r \in R(\sigma) \land w \in W(\sigma) \land \mathsf{vis}(w) \land \neg\exists w'.\ (w', r) \in \mathsf{wr}\}$.*

A dependency graph $G = (V(\sigma), \mathsf{vis}, \mathsf{rt}(\sigma), \mathsf{wr}, \mathsf{ww}, \mathsf{rw})$ consists of a set of vertices $V(\sigma)$ and edges $\mathsf{rt}(\sigma) \cup \mathsf{wr} \cup \mathsf{ww} \cup \mathsf{rw}$. To prove linearizability, we rely on the following theorem.

Theorem 4.12. *An execution σ is linearizable if there exist vis, wr, ww, and rw such that $G = (V(\sigma), \mathsf{vis}, \mathsf{rt}(\sigma), \mathsf{wr}, \mathsf{ww}, \mathsf{rw})$ is an acyclic dependency graph.*

Proof. Let $R'(\sigma)$ be the set of incomplete read operations in the execution σ and let $V'(\sigma) = \{o \mid o \in V(\sigma) \wedge \mathsf{vis}(o)\} \setminus R'(\sigma)$. Since G is acyclic, the subgraph G' of G induced by $V'(\sigma)$ is also acyclic. Thus, the partial order over $V'(\sigma)$ induced by G' can be extended to a total order $\prec$ such that for every edge $(o_1, o_2) \in G'$, we have $o_1 \prec o_2$. We now argue that the history π, obtained by ordering operations in $V'(\sigma)$ according to $\prec$ and completing incomplete writes, is a linearization of σ. We first prove that π is well-formed:

- All complete operations are visible, so π contains all complete operations.
- If $(o_1, o_2) \in \mathsf{rt}(\sigma)$, then $o_1 \prec o_2$, so π preserves the real-time order, i.e., $\mathsf{rt}(\sigma) \subseteq \pi$.
- π is well-formed in that only finitely many operations can precede any complete operation o: since o completes in σ, only finitely many operations can start before it returns. Let $\mathcal{S}$ be the set of such operations. Every operation in $V'(\sigma) \setminus \mathcal{S}$ starts after o returns in σ, so o precedes it in real time. Since $\mathsf{rt}(\sigma) \subseteq \pi$, only operations in $\mathcal{S}$ can precede o in $\prec$ and therefore in π.
- The same holds for incomplete operations. The definition of $V'(\sigma)$ implies that an incomplete operation can only be a visible write. By Definition 4.11 – 2 – (iv), for any visible write w, there exists a complete read r such that $(w, r) \in \mathsf{wr}$. It follows that $w \prec r$, so w precedes r in π. Since r has only finitely many predecessors in π, the same holds for w.

Finally, we show that π satisfies the semantics of a register. That is, for any read operation r in π, either there exists a latest write operation w that precedes r in π and $\mathsf{val}(w) = \mathsf{val}(r)$, or no such w exists and $\mathsf{val}(r) = \bot$. Assume by contradiction that such a w exists but $\mathsf{val}(w) \neq \mathsf{val}(r)$. There are two cases:

- If $\mathsf{val}(r) = \bot$, then since no $w' \in W(\sigma)$ has $\mathsf{val}(w') = \bot$, there is no $w' \in W(\sigma)$ such that $(w', r) \in \mathsf{wr}$. By definition of rw, for all visible w', we must have $(r, w') \in \mathsf{rw}$, hence $r \prec w$. This contradicts the assumption that w precedes r in π.
- If $\mathsf{val}(r) \neq \bot$, then by Definition 4.11 – 2 – (iii) and (i), there exists $w' \in W(\sigma)$ such that $\mathsf{vis}(w')$, $\mathsf{val}(w') = \mathsf{val}(r)$, and $(w', r) \in \mathsf{wr}$, implying $w' \prec r$. Since $\mathsf{vis}(w)$, $\mathsf{vis}(w')$ and ww is a total order over all the visible writes, we have that $(w, w') \in \mathsf{ww}$ or $(w', w) \in \mathsf{ww}$. If $(w, w') \in \mathsf{ww}$, then $w \prec w' \prec r$, contradicting that w is the latest write before r in π. If instead $(w', w) \in \mathsf{ww}$, then by the definition of rw we must have $(r, w) \in \mathsf{rw}$, so $r \prec w$, contradicting that w precedes r in π.

Now assume that no such w exists. We must show that $\mathsf{val}(r) = \bot$. Assume by contradiction that $\mathsf{val}(r) \neq \bot$. Then, by Definition 4.11 – 2 – (iii), there exists $w' \in W(\sigma)$ such that $\mathsf{vis}(w')$ and $(w', r) \in \mathsf{wr}$. It follows that $w' \prec r$, so w' precedes r in π. Thus there exists a latest write preceding r in π, contradicting the assumption that no such w exists. □

We now prove that every execution of the protocol in Figure 4.5 is linearizable. Fix one such execution σ. Our strategy is to find witnesses for vis, wr, ww and rw that validate the conditions of Theorem 4.12. To this end, consider the function $\tau : \sigma \to (\mathbb{N} \cup \{\infty\}) \times \mathbb{N}$ that maps each operation $o \in V(\sigma)$ to a version as follows:

- If $o \in R(\sigma)$ and o executes line 10 in Figure 4.5, then $\tau(o) = s'.\mathsf{ver}$.
- If $o \in W(\sigma)$ and o executes line 5 in Figure 4.5, then $\tau(o) = t$.
- Otherwise, $\tau(o) = (\infty, i)$, where p_i is the process that executes o.

We then define the required witnesses as follows:

- $\mathsf{vis}(o)$ is true if o completes or if $o \in W(\sigma)$ and there exists a complete read $r \in R(\sigma)$ such that $\tau(o) = \tau(r)$;
- $(w, r) \in \mathsf{wr}$ if and only if $w \in W(\sigma)$, $r \in R(\sigma)$, r completes, and $\tau(w) = \tau(r)$;
- $(w, w') \in \mathsf{ww}$ if and only if $w, w' \in W(\sigma)$, $\mathsf{vis}(w)$, $\mathsf{vis}(w')$, and $\tau(w) < \tau(w')$; and
- rw is derived from wr and ww as per the dependency graph definition.

Our proof relies on the next proposition. We omit its easy proof which follows from the $\mathsf{Validity}$ property of the quorum access functions:

Proposition 4.13. *The following hold:*

1. *For every $w_1, w_2 \in W(\sigma)$, $\tau(w_1) = \tau(w_2)$ implies $w_1 = w_2$.*
2. *For every $w \in W(\sigma)$, $\tau(w) > (0, 0)$.*
3. *For every $r \in R(\sigma)$ that completes, either $\tau(r) = (0, 0)$ or there exists $w \in W(\sigma)$ such that $\mathsf{vis}(w)$ and $\tau(r) = \tau(w)$.*
4. *For every $r \in R(\sigma)$ and $w \in W(\sigma)$, $\tau(r) = \tau(w)$ implies $\mathsf{val}(r) = \mathsf{val}(w)$.*

Our proof also relies on the following auxiliary lemma:

Lemma 4.14. *The following hold:*

1. *For all $r, w \in V(\sigma)$, if $(r, w) \in \mathsf{rw}$ then $\tau(r) < \tau(w)$.*
2. *For all $o_1, o_2 \in V(\sigma)$, if $(o_1, o_2) \in \mathsf{rt}(\sigma)$, then $\tau(o_1) \leq \tau(o_2)$. Moreover, if o_2 is a* write, *then $\tau(o_1) < \tau(o_2)$.*

Proof. We prove each item separately:

1. Let $r, w \in V(\sigma)$ be such that $(r, w) \in \mathsf{rw}$. There are two cases:
    - Suppose that for some w' we have $(w', r) \in \mathsf{wr}$ and $(w', w) \in \mathsf{ww}$. The definition of wr implies that $\tau(r) = \tau(w')$, and the definition of ww implies that $\tau(w') < \tau(w)$. Then $\tau(r) < \tau(w)$.
    - Suppose now that $\neg\exists w'.\ (w', r) \in \mathsf{wr}$. We show that $\tau(r) = (0, 0)$. Indeed, if $\tau(r) \neq (0, 0)$, then by Proposition 4.13.3, there exists $w \in V(\sigma)$ such that $\tau(r) = \tau(w)$. But then $(w, r) \in \mathsf{wr}$, contradicting the assumption that there is no such write. At the same time, Proposition 4.13.2 implies that $\tau(w) > (0, 0)$. Then $\tau(r) < \tau(w)$.
2. Let $o_1, o_2 \in V(\sigma)$ be such that $(o_1, o_2) \in \mathsf{rt}(\sigma)$. Because o_1 precedes o_2, o_1 completes.

Let u be the update function computed during o_1's invocation at lines 6 (if o_1 is a write) or line 11 (if o_1 is a read). The definitions of u imply that right after a process applies it to its state it has $\mathsf{ver} \geq \tau(o_1)$.

Let p_i be the process that invokes o_2. Suppose the $\mathsf{quorum\_get}$ invocation by o_2 does not complete. Then, by the definition of τ, we have $\tau(o_2) = (\infty, i)$, and thus $\tau(o_1) < \tau(o_2)$, as required. Otherwise, let $S = \{s_i \mid i = 1..k\}$ be the states returned by the corresponding $\mathsf{quorum\_get}()$ invocation (lines 3 and 9). The Validity property of the quorum access functions ensures that for each $s_i \in S$ there exists a set of previous invocations $\{\mathsf{quorum\_set}(u^i_j) \mid j = 1..k_i\}$ such that s_i is the result of applying the update functions in $\{u^i_j \mid j = 1..k_i\}$ to the initial state in some order. Since the $\mathsf{quorum\_get}()$ invocation happens after $\mathsf{quorum\_set}(u)$ has completed, by the Real-time ordering property there is at least one state s_r to which u has been applied: $u = u^r_j$ for some j. Because the update functions passed by the protocol to $\mathsf{quorum\_set}()$ never decrease ver, we then have $s_r.\mathsf{ver} \geq \tau(o_1)$. There are two cases:

- Suppose o_2 is a write. Because $\tau(o_2)$ is greater than the maximum ver value among the states in S (lines 4-5), we have $\tau(o_1) < \tau(o_2)$.
- Suppose o_2 is a read. Because $\tau(o_2)$ is the maximum ver value among the states in S (line 10), we have $\tau(o_1) \leq \tau(o_2)$. □

Theorem 4.15. $G = (V(\sigma), \mathsf{vis}, \mathsf{rt}(\sigma), \mathsf{wr}, \mathsf{ww}, \mathsf{rw})$ *is an acyclic dependency graph.*

Proof. From Proposition 4.13 and the definitions of vis, wr, ww and rw it follows that G is a dependency graph. We now show that G is acyclic. By contradiction, assume that the graph G contains a cycle $o_1, \dots, o_n = o_1$. Then $n > 1$. By Lemma 4.14 and the definitions of τ, ww and wr, we must have $\tau(o_1) \leq \dots \leq \tau(o_n) = \tau(o_1)$, so that $\tau(o_1) = \dots = \tau(o_n)$. Furthermore, if (o, o') is an edge of G and o' is a write, then $\tau(o) < \tau(o')$. Hence, all the operations in the cycle must be reads, and thus, all the edges in the cycle come from $\mathsf{rt}(\sigma)$. Then there exist reads r_1, r_2 in the cycle such that r_1 completes before r_2 is invoked and r_2 completes before r_1 is invoked, which is a contradiction. □

4.4 Bounded Process Failures

The upper bound presented in §4.3 is correct under very general conditions and answers a fundamental computability question. However, its progress relies on processes from some read quorum asynchronously and independently bumping their clocks to reach the desired cut-off. As a result, the time it takes for an operation to complete may grow arbitrarily large, even under favourable network conditions.

To address this issue, we turn our attention to a practically motivated class of fail-prone systems in which at most a minority of processes may fail. While this assumption narrows the set of admissible failure patterns, it yields stronger connectivity guarantees that we leverage to design a more efficient implementation – one where operation latency can be bounded by message delays and state exchange rates. This implementation also uses bounded memory,

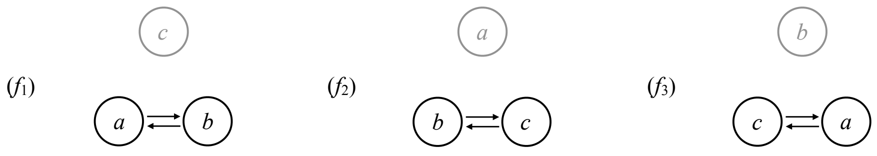

Figure 4.6: A 1-fail-prone system $\mathcal{F}$ consisting of failure-patterns f_i, $i = 1..3$.

reflecting practical system considerations. The resulting protocol is therefore better suited for practical use.

4.4.1 Lower Bounds

When there are no restrictions on the fail-prone system, a straightforward lifting of Lemma 4.1 yields the following:

Corollary 4.16. *Let $\mathcal{F}$ be a fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ a termination mapping. If some algorithm $\mathcal{A}$ is an $(\mathcal{F}, \tau)$-obstruction-free implementation of lattice agreement over $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), then for all $f \in \mathcal{F}$, $\tau(f)$ is strongly connected in $\mathcal{G} \setminus f$.*

However, when the number of process failures is bounded, stronger connectivity guarantees emerge. To formalize this, we introduce the following notion. A *k-fail-prone system $\mathcal{F}$* allows any set of k processes or fewer to fail, but disallows the failure of more than k processes:

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

Example 4.17. Consider the set of processes $\mathcal{P} = \{a, b, c\}$. In Figure 4.6, we depict a 1-fail-prone system consisting of failure patterns f_i, $i = 1..3$. Under the failure pattern f_1, processes a and b are correct while c may crash. The channels (a, b) and (b, a) are correct, while all others may fail.

Example 4.18. The system $\mathcal{F}_M$ from §3.2 is $\lfloor \frac{n-1}{2} \rfloor$-fail-prone.

The following theorem establishes the minimal connectivity required to implement lattice agreement, atomic snapshots, or atomic registers over the model $\mathcal{M}_{\text{ARD}}$ (asynchronous / reliable / disconnected), in systems where any set of k or fewer processes may fail. Like our general lower bound (Theorem 3.7), this result assumes a weak termination guarantee: obstruction-freedom is required to hold only at some non-empty set of processes for each failure pattern. The theorem states that, regardless of how small this set is, it must be contained in a group of more than k correct processes that are strongly connected by correct channels.

Theorem 4.19. *Let $\mathcal{F}$ be a k-fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ a termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) \neq \emptyset$. Assume that there exists an $(\mathcal{F}, \tau)$-obstruction-free implementation of any of the following objects over the model $\mathcal{M}_{\text{ARD}}$: MWMR atomic registers,* *SWMR atomic snapshots, or lattice agreement. 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.*

As before, we prove Theorem 4.19 for lattice agreement. Since SWMR atomic snapshots can be constructed from MWMR atomic registers (§2.6.3), and lattice agreement can in turn be built from snapshots (§2.6.4), the lower bounds for the former two problems follow.

Proof of Theorem 4.19 for lattice agreement. The result follows from Theorem 3.7 together with the following claim: if a k-fail-prone system $\mathcal{F}$ admits a generalized quorum system, then for every $f \in \mathcal{F}$, f-available write quorums must have size strictly greater than k.

Suppose, for contradiction, that there exists a failure pattern $f \in \mathcal{F}$ such that some f-available write quorum has size at most k. Let W be one such quorum. Since $\mathcal{F}$ is a k-fail-prone system and $|W| \leq k$, there exists a failure pattern $g \in \mathcal{F}$ such that every process in W is faulty in g. By the Availability property of generalized quorum systems, there must exist a read quorum R such that every process in R is correct in g. Hence $W \cap R = \emptyset$, contradicting the Consistency property of generalized quorum systems. Therefore, every f-available write quorum W must satisfy $|W| > k$.

Finally, by Theorem 3.7 we have that for every $f \in \mathcal{F}$ and every f-available write quorum W, $W \subseteq U_f$. Hence $|U_f| > k$ and $\tau(f) \subseteq U_f$. □

Theorem 4.19 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 [49] and partially synchronous consensus [32]. 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 that there can exist at most one connected core for a given $\mathcal{G}$ and f.

Example 4.20. Consider the 1-fail-prone system from Figure 4.6. For the failure pattern f_1 the connected core is $\{a, b\}$, for the failure pattern f_2 the connected core is $\{b, c\}$, and for the failure pattern f_3 the connected core is $\{c, a\}$.

We now demonstrate that the restricted class of fail-prone systems considered in this section induces connectivity requirements that correspond to a particular instance of a generalized quorum system, thereby ensuring that our results remain consistent with the general framework. Specifically, we show that the existence of a connected core for each failure pattern yields such a quorum system.

For each $f \in \mathcal{F}$, let $\mathcal{S}_f$ denote the connected core guaranteed by Theorem 4.19 – a set of more than k correct processes that are strongly connected by correct channels under f. Define $\mathcal{R} = \{\mathcal{S}_f \mid f \in \mathcal{F}\}$ and $\mathcal{W} = \{\mathcal{S}_f \mid f \in \mathcal{F}\}$. Then $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ forms a generalized quorum system: since each $\mathcal{S}_f$ contains a majority, read and write quorums intersect; and its strong connectivity under f ensures that it is both f-available and f-reachable from itself. Hence, the Consistency and Availability conditions of a GQS are satisfied.

4.4.2 Upper bounds

We now describe a protocol for implementing atomic registers in the model $\mathcal{M}_{\text{AEF}}$, under the assumption that any minority of processes may fail. The protocol is efficient in that it ensures bounded latency, bounded memory usage, and bounded message complexity per operation.

The design follows the modular construction presented in Figure 4.5, which separates the logic of register emulation from the mechanism used to access quorums. In particular, it assumes the existence of abstract quorum access functions – $\mathsf{quorum\_get}()$ and $\mathsf{quorum\_set}(u)$ – that retrieve and propagate state, respectively. This construction already guarantees the desired efficiency properties *modulo* the implementation of the quorum access functions. Thus, to obtain an efficient protocol for atomic registers, it suffices to implement quorum access with the same guarantees.

To this end, in Figure 4.7 we present a protocol that leverages the existence of a connected core in every execution to achieve the desired efficiency bounds. Unlike §4.3, we do not assume that the connectivity graph induced by correct processes and reliable channels is transitively connected – since we are concerned with practicality, we must account for this, as dissemination across multiple hops can become a dominant cost in efficiency. To handle this, the protocol relies on repeated gossip to disseminate state. Despite this, it uses only bounded metadata, and its progress depends solely on message delays and the rate of state exchange – making it better suited for practical use than the general upper bound of §4.3.

State and communication. Like in §4.3, 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: it can only be manipulated through update functions passed by the callers. Each process also maintains a monotonically increasing sequence number seq (initially 0), which is used to generate a unique identifier for each quorum access invocation. This identifier is included in every metadata entry, allowing processes to associate messages with the correct operation.

To track the state of quorum gets and sets across the system, each process maintains the following bounded metadata:

- $\mathsf{get\_req}[i]$ stores the most recent get request issued by p_i, tagged with seq;
- $\mathsf{get\_resp}[i][j]$ records p_i's response to p_j's latest get request;
- $\mathsf{set\_req}[i]$ stores the most recent set request issued by p_i, as a pair (seq, u), where u is the corresponding update function; and
- $\mathsf{set\_resp}[i][j]$ records p_i's acknowledgement of p_j's latest set request.

For simplicity, the pseudocode in Figure 4.7 separates computation from communication. Most handlers update metadata arrays locally, without sending messages. Instead, each process periodically sends a single `STATE` message (line 26) that includes all relevant metadata.

Upon receiving a `STATE` message (line 27), a process merges the received metadata into its own (lines 28 – 37), keeping only the most recent entries based on sequence numbers. This ensures that information is disseminated across the system using only bounded memory, even

```
seq ← $0$
state $\in \mathcal{S}$    // opaque state of the top-level protocol
function quorum_get():
    pre: status = IDLE
    seq ← seq + $1$
    get_req[$i$] ← seq
    status ← GET_QUERY
    async wait until status = GET_DONE
    return {get_resp[$j$][$i$].$val$ | get_resp[$j$][$i$].$seq$ = seq}
when $\exists j$. get_req[$j$].$seq$ > get_resp[$i$][$j$].$seq$
    get_resp[$i$][$j$] ← (get_req[$j$].$seq$, state)
when status = GET_QUERY ∧ |{$j$ | get_resp[$j$][$i$].$seq$ = seq}| > $\frac{n}{2}$
    status ← GET_DONE
function quorum_set($u$):
    pre: status = IDLE
    seq ← seq + $1$
    set_req[$i$] ← (seq, $u$)
    status ← SET_QUERY
    async wait until status = SET_DONE
when $\exists j$. set_req[$j$].$seq$ > set_resp[$i$][$j$].$seq$
    state ← set_req[$j$].$u$(state)
    set_resp[$i$][$j$] ← set_req[$j$].$seq$
when status = SET_QUERY ∧ |{$j$ | set_resp[$j$][$i$].$seq$ = seq}| > $\frac{n}{2}$
    status ← SET_DONE
periodically
    send STATE(get_req, get_resp, set_req, set_resp) to all
when received STATE($G_{\text{req}}, G_{\text{resp}}, S_{\text{req}}, S_{\text{resp}}$)
    for $p_j \in \mathcal{P}$ do
        if $G_{req}[j].seq$ > get_req[$j$].$seq$ then
            get_req[$j$] ← $G_{\text{req}}[j]$
        if $S_{req}[j].seq$ > set_req[$j$].$seq$ then
            set_req[$j$] ← $S_{\text{req}}[j]$
        for $p_k \in \mathcal{P}$ do
            if $G_{resp}[j][k].seq$ > get_resp[$j$][$k$].$seq$ then
                get_resp[$j$][$k$] ← $G_{\text{resp}}[j][k]$
            if $S_{resp}[j][k].seq$ > set_resp[$j$][$k$].$seq$ then
                set_resp[$j$][$k$] ← $S_{\text{resp}}[j][k]$
```

Figure 4.7: Quorum access functions under bounded process failures: the protocol at a process p_i.

when communication proceeds across multiple hops due to indirect connectivity. The periodic propagation of state also copes with message loss on channels that are only eventually reliable.

Protocol operation. Upon a call to quorum_get() (line 3), the process increments its local sequence number, records the request in get_req$[i]$, and sets status = GET_QUERY (lines 5–7). This request is disseminated to all processes via periodic STATE messages (line 26). When another process p_j receives a fresher request from p_i (line 10), it responds by recording its current local state in get_resp$[j][i]$. These responses are eventually propagated back to p_i through gossip and merged as part of the handler in lines 28–37. As responses accumulate, p_i checks whether a quorum of responses match its current sequence number (line 12). Once this occurs, it sets status = GET_DONE (line 13) and returns the set of states retrieved from the quorum (line 9).

Upon a call to quorum_set(u) (line 14), the process increments its local sequence number, stores the update function in set_req$[i]$, and sets status = SET_QUERY (lines 16–18). This intent to perform an update is gossiped to all processes via periodic STATE messages (line 26). When another process p_j receives a fresher request from p_i (line 20), it applies the update function to its local state (line 21) and records the acknowledgement in set_resp$[j][i]$ (line 22). These acknowledgements are eventually delivered back to p_i through gossip and merged in the handler in lines 28–37. As acknowledgements accumulate, p_i monitors the number of responses matching its current sequence number (line 23). Once it observes a quorum, it sets status = SET_DONE (line 24) and returns from the call.

Each process stores at most one metadata entry per process and per operation, and no unbounded buffering is required. All communication occurs via bounded-size STATE messages. This design guarantees bounded memory usage, bounded message complexity, and progress that depends solely on the rate at which messages are delivered and exchanged – thereby meeting the efficiency goals outlined at the start of this section.

Correctness. The proof that this protocol satisfies Validity and Real-time ordering follows the same structure as for the quorum access functions in classical quorum systems (see §4.3.1). Therefore, we focus here on proving liveness.

Lemma 4.21. *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 for each $f \in \mathcal{F}$, $\tau(f) = \mathcal{S}_f$. Then the algorithm in Figures 4.7 is $(\mathcal{F}, \tau)$-wait-free.*

Proof. We prove the liveness of quorum_get(); the case of quorum_set(u) is analogous. Fix a failure pattern $f \in \mathcal{F}$ and a process $p_i \in \tau(f) = \mathcal{S}_f$ that invokes quorum_get(). By assumption, $\mathcal{S}_f$ is a set of correct processes that are strongly connected by correct channels under f.

Upon invoking quorum_get(), the process p_i increments its local sequence number and sets status = GET_QUERY (lines 5–7). This request is then gossiped periodically to all processes (line 26). Since the processes in $\mathcal{S}_f$ are strongly connected under f and $p_i \in \mathcal{S}_f$, every member of $\mathcal{S}_f$ will eventually receive p_i's updated get_req$[i]$. Whenever a process $p_j \in \mathcal{S}_f$ observes that get_req$[i]$ carries a higher sequence number than previously seen (line 10), it records its current local state in get_resp$[j][i]$. This metadata is then propagated back to p_i via periodic STATE messages, which are merged upon receipt (lines 28–37). Since gossip is repeated and all channels between members of $\mathcal{S}_f$ are eventually reliable under f, p_i will eventually receive responses from all members of a majority quorum within $\mathcal{S}_f$ with matching sequence numbers. This is guaranteed because p_i does not increment seq or begin a new quorum access operation until the current one has completed. At that point, the guard in line 12 is satisfied, and p_i sets status $=$ GET_DONE (line 13), allowing the operation to complete. $\square$

Chapter 5

Consensus

In this chapter, we study the implementation of consensus in partially synchronous systems subject to arbitrary process and channel failures. We show that a generalized quorum system is also a tight reliability bound for consensus. This result complements the bounds established for atomic registers, atomic snapshots and lattice agreement in the asynchronous setting.

In §5.1, we first establish that a generalized quorum system is a lower bound. To this end, we show how consensus can be used to implement a MWMR atomic register in a partially synchronous system. We then reduce the lower bound for consensus to the lower bound for registers proved in §4.1.

Then, in §5.2, we present an upper bound that matches this lower bound. Assuming the existence of a generalized quorum system, we give a consensus algorithm that guarantees wait-freedom within its write quorums. Similar to the assumptions in the previous chapter, we consider an adversarial model where correct channels are only eventually timely and faulty channels are flaky, meaning they may drop any number of messages without fairness guarantees. In this setting, consensus is simpler to implement than registers: a process can leverage the eventual synchrony of the network to determine whether the information it receives is up to date. This property is crucial for safety but difficult to guarantee in the asynchronous model.

Having established tight bounds for solvability, we then turn – as in the previous chapter – to a practically motivated class of fail-prone systems in which at most a minority of processes may fail (§5.3). We prove that if consensus is solvable under an $\frac{n}{2}$-fail-prone system, then a connected core must exist in every execution. The key challenge then becomes leveraging this stronger connectivity guarantee to design a more efficient implementation of consensus – one where operation latency can be bounded by message delays and state exchange rates. Furthermore, we require this implementation to use bounded memory and guarantee wait-freedom within the connected core.

As explained in Chapter 1, classical abstractions such as failure or leader detectors (see §2.7) lose their utility under flaky channels, since message loss renders these algorithms inapplicable. We therefore take a different approach and adapt the recently proposed abstraction of a *view synchronizer* [17, 18, 54, 55], which generalizes these mechanisms and enables coordination despite arbitrary message loss. At a high level, the synchronizer aims to gather a sufficiently large set of well-connected correct processes into a view led by a well-connected correct leader, and to keep them in that view long enough to drive the protocol to completion. We present its specification (§5.4.1), an implementation (§5.4.2), and prove that this implementation meets the specification under partial synchrony (§5.4.3).

We then present a modular consensus protocol (§5.4.4) that leverages the synchronizer to coordinate processes within the connected core and achieves the desired efficiency and termination guarantees. As in §4.4.1, the algorithm explicitly accounts for propagation of information and does not assume that the connectivity graph is transitively connected.

Finally, in §5.5, we demonstrate how our lower and upper bounds can be applied to establish consensus solvability in various existing models of weak connectivity [5, 6, 33, 43, 51]. In particular, we show that some connectivity models strong enough to implement the Ω leader detector are nevertheless too weak to solve consensus.

5.1 Lower Bound

We now argue that Theorem 3.10 holds. The proof of the lower bound for lattice agreement (§4.1) remains valid in the partially synchronous model $\mathcal{M}_{\text{PRD}}$: since the execution σ constructed in that proof is finite, it is also valid under partial synchrony, where all actions occur before GST. As in the asynchronous case, the lower bound for lattice agreement implies the one for registers under partial synchrony [2, 14]. Since, as we show next, consensus can be used to implement a MWMR atomic register under partial synchrony, the lower bound for consensus follows directly from the lower bound for registers.

5.1.1 Registers from Consensus

It is well-known that wait-free atomic registers can be implemented from wait-free consensus in the classical crash-stop model with reliable channels. We now show that a similar result holds for obstruction-freedom in the model $\mathcal{M}_{\text{PRD}}$ (partially synchronous / reliable / disconnected).

In Figure 5.1, we present an algorithm that achieves this by using an infinite array of obstruction-free consensus instances. Each operation – read or write – is encoded as a command and proposed sequentially in these instances until it is chosen. Both operations terminate at the first instance where their command is decided. In this way, processes agree on a total order of operations, which defines the sequence of commands applied to the register.

Theorem 5.1. *Let $\mathcal{F}$ be a fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be a termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) \neq \emptyset$. Then the algorithm in Figure 5.1 is an $(\mathcal{F}, \tau)$-obstruction-free implementation of a MWMR atomic register over $\mathcal{M}_{\text{PRD}}$.*

Proof. It is straightforward to verify that the algorithm implements a MWMR atomic register. Therefore, we focus on proving obstruction-freedom. Let $f \in \mathcal{F}$, σ be an f-compliant execution of the algorithm in Figure 5.1, $p \in \tau(f)$, and *arg_op* be an operation invoked by p that executes solo in σ. We show that *arg_op* must eventually return.

```
idx ← $0$
val ← $\bot$
let $C$ be an infinite array of $(\mathcal{F}, \tau)$-obstruction-free consensus instances over $\mathcal{M}_{\text{PRD}}$
upon invocation of $arg\_op$
  pre: op $= \bot \wedge arg\_op \neq \bot$
  op ← $arg\_op$
  while TRUE do
    let $op = C$[idx].propose(op)
    idx ← idx $+ 1$
    if $op =$ write($v$) then
      val ← $v$
    if $op =$ op then
      op ← $\bot$
      if $op =$ read then $ret\_val$ ← val
      else $ret\_val$ ← ACK
      return $ret\_val$ in response to op
```

Figure 5.1: Protocol at a process p_i. Let Value be an arbitrary domain of values. The operation arg_op can either be read() or write(v) for $v \in$ Value.

Let i be such that arg_op is first proposed in the i-th instance of consensus at a time t, and assume, for the sake of contradiction, that arg_op never returns. Since arg_op executes solo in σ, it is not concurrent with any operation invoked by another correct process. Because arg_op never returns, it follows that no operation invocation by a correct process occurs at any time $\geq t$. Hence, from time t onwards, the only outstanding operations are either arg_op itself or operations invoked by faulty processes. Therefore, every proposal by p at a time $\geq t$ executes solo in σ.

We now show that for each $j \geq i$, arg_op is proposed by p in the j-th instance of consensus $p.C[j]$. Since i is defined as the first instance of consensus where arg_op is proposed by p, the claim trivially holds for $j = i$. Suppose now that arg_op is proposed in the k-th instance of consensus for some $k > i$. Because the k-th instance of consensus is $(\mathcal{F}, \tau)$-obstruction-free and $C[k]$.propose(arg_op) executes solo in σ, the operation $C[k]$.propose(arg_op) must eventually return with some value $op \neq arg\_op$. Consequently, the handler at line 4 ensures that arg_op is proposed by p in the $(k+1)$-th instance of consensus, as required.

Therefore, there exists $r \geq i$ such that p invokes $C[r]$.propose(arg_op) after all faulty processes have crashed, so that p is the only process to propose in $C[r]$. Since the r-th instance of consensus is $(\mathcal{F}, \tau)$-obstruction-free and $C[r]$.propose(arg_op) executes solo in σ, $C[r]$.propose(arg_op) must eventually return some value $op \neq arg\_op$. However, because p is the only process to propose in $C[r]$ and the decided value must come from a proposal, this contradicts the safety property of consensus. Hence, our assumption was false and arg_op must eventually return in σ. □

```
view, aview ← $0, 0$
val, my_val ← $\bot, \bot$
phase ∈ {ENTER, PROPOSE, ACCEPT, DECIDE}
**function** propose($x$):
    my_val ← $x$
    **async wait until** phase = DECIDE
    **return** val
**when received** $\{$1B(view, $v_j, x_j$) $\mid p_j \in R\}$ **from some** $R \in \mathcal{R}$
    **pre:** phase = ENTER
    **if** $\forall p_j \in R.\ x_j = \bot$ **then**
        **if** my_val = $\bot$ **then return**
        **send** 2A(view, my_val) **to all**
    **else**
        **let** $p_j \in R$ **be such that** $x_j \neq \bot \wedge$
          $(\forall p_k \in R.\ x_k \neq \bot \implies v_k \leq v_j)$
        **send** 2A(view, $x_j$) **to all**
    phase ← PROPOSE
**when received** 2A(view, $x$)
    **pre:** phase ∈ {ENTER, PROPOSE}
    val ← $x$
    aview ← view
    **send** 2B(view, $x$) **to all**
    phase ← ACCEPT
**when received** $\{$2B(view, $x$) $\mid p_j \in W\}$ **from some** $W \in \mathcal{W}$
    val ← $x$
    aview ← view
    phase ← DECIDE
**on startup or when the timer** view_timer **expires**
    view ← view + 1
    start_timer(view_timer, view $\cdot C$)
    **send** 1B(view, aview, val) **to** leader(view)
    phase ← ENTER
```

Figure 5.2: Consensus protocol at a process p_i. C is any positive constant.

5.2 Upper Bound

We now present a consensus protocol in the model $\mathcal{M}_{\text{PEF}}$ (partially synchronous / eventually reliable / flaky) that validates Theorem 3.11. To this end, we fix a generalized quorum system $(\mathcal{F}, \mathcal{R}, \mathcal{W})$ and a termination mapping $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ such that $\tau(f) = U_f$ holds for each $f \in \mathcal{F}$. As in §4.3, 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 given in Figure 5.2.

Consensus vs registers under channel failures. Interestingly, solving consensus under process and channel failures is simpler than implementing registers. The main challenge in implementing registers was determining whether the information received by a process is up to date (§4.3). This is particularly difficult in the asynchronous model with unidirectional connectivity, where processes cannot rely on bidirectional exchanges to confirm freshness. In the partially synchronous model, however, processes can exploit the eventual timeliness of the network to make this determination. Technically, this is achieved using a *view synchronizer* [11, 54], explained next. Then the connectivity stipulated by a generalized quorum system is sufficient to implement $(\mathcal{F}, \tau)$-wait-free consensus using an algorithm similar to Paxos [47].

View synchronization. As observed in Chapter 1, it can be hard or even impossible to implement Ω (§2.7.2) in the presence of channel failures. Intuitively, the difficulty is that an implementation may be unable to reliably identify processes with sufficiently robust connectivity, which is necessary to ensure the liveness of consensus. To address this, we construct our solution by combining the classical single-decree Paxos algorithm [47] with the communication-less view synchronizer [11, 54]. Our key observation is that this combination remains correct and ensures progress even under the limited process and channel reliability guarantees provided by $(\mathcal{F}, \mathcal{R}, \mathcal{W})$.

The consensus protocol works in a succession of views, each with a designated leader process $\mathsf{leader}(v) = p_{((v-1) \bmod n)+1}$. 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 [11, 54]. 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 $\mathsf{view\_timer}$ 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 5.2. *Let $d > 0$ be arbitrary. There exists a view $\mathcal{V}$ such that for every $v \geq \mathcal{V}$, all correct processes overlap in v for at least d time units.*

Proof. Let $D, D' \geq 0$ be such that after GST, no correct process is ahead of real time by more than D and none is behind by more than D'. For a view v, let e_v and l_v denote the real times at which a process would enter and leave v, respectively, if its clock did not drift:

$$e_v = \sum_{i=0}^{v-1} i\,C = \frac{(v-1)v}{2}\,C \qquad \text{and} \qquad l_v = e_v + v\,C.$$

After GST, any correct process enters v no later than $e_v + D'$ and leaves v no earlier than $l_v - D$. Hence, every correct process is in v throughout the interval $[\,e_v + D',\, l_v - D\,]$, whose length is

$$(l_v - D) - (e_v + D') = (l_v - e_v) - (D + D') = v\,C - (D + D').$$

Choose $\mathcal{V}$ large enough that *(i)* $e_{\mathcal{V}} \geq \mathsf{GST}$ and *(ii)* $\mathcal{V}\,C > d + D + D'$. Then for every $v \geq \mathcal{V}$ we have $v\,C \geq \mathcal{V}\,C > d + D + D'$, so the common-overlap interval $[\,e_v + D',\, l_v - D\,]$ has length strictly greater than d. Therefore, all correct processes overlap in every view $v \geq \mathcal{V}$ for at least d. □

Protocol operation. A process stores its initial proposal in my_val, the last proposal it accepted in val, and the view in which this happened in aview. A variable phase tracks the progress of the process through the phases of the protocol. Upon entering a view v, the process sends its last-accepted value to leader(v) in an 1B message (line 30). This message is analogous to Paxos's 1B; there is no analog of 1A, since leader election is handled by the synchronizer.

A leader waits to collect 1B messages from all processes in some read quorum for its view (line 8); messages from lower views (out of date) are ignored. From these messages, the leader computes its proposal as in Paxos: if any process has previously accepted a value, the leader selects the one accepted in the maximal view; if no such value exists and propose() has already been invoked at the leader, it proposes its own value; otherwise, it skips its turn.

A process awaits a 2A message from the leader of its view (line 17) and, upon receipt, accepts by updating val and aview, then broadcasts a 2B message. When a process receives matching 2B messages from all processes in some write quorum (line 23), it concludes that a decision has been reached and sets phase = DECIDE. If a propose() is pending, this satisfies the condition at line 6, and the process returns the decided value to the caller.

Proof of Theorem 3.11. 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 $(\mathcal{F}, \tau)$-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 3.6, $W \subseteq U_f$. 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 5.2 and since leaders rotate round-robin, there exists a view v led by p such that all correct processes enter v after $\max(\mathsf{GST}, t')$ 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. □

5.3 Lower Bound under Bounded Process Failures

The lower bounds in §4.4.1 also apply to consensus under partial synchrony, that is, in the model $\mathcal{M}_{\text{PRD}}$ (partially synchronous / reliable / disconnected), with analogs of Lemma 4.1 and Theorem 4.19 stated as follows:

Lemma 5.3. *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 $\mathcal{M}_{\text{PRD}}$, then T is strongly connected in $\mathcal{G} \setminus f$.*

Theorem 5.4. *Let $\mathcal{F}$ be a k-fail-prone system and $\tau : \mathcal{F} \to 2^{\mathcal{P}}$ be a termination mapping such that for each $f \in \mathcal{F}$, $\tau(f) \neq \emptyset$. Assume that there exists an $(\mathcal{F}, \tau)$-obstruction-free implementation of consensus over $\mathcal{M}_{\text{PRD}}$. 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 results hold, observe first that Lemma 4.1 and Theorem 4.19 also remain valid if the model $\mathcal{M}_{\text{ARD}}$ is replaced with $\mathcal{M}_{\text{PRD}}$: since the executions σ constructed in their proofs are finite, they are also valid executions under $\mathcal{M}_{\text{PRD}}$, where all actions occur before GST. The claim then follows from Theorem 5.1, which states that registers can be implemented from consensus.

While the solution presented in §5.2 establishes a tight upper bound and resolves a fundamental solvability question, the synchronizer it relies on – though simple and well-known – is inefficient, incurring high latency. This limits its practicality, especially in settings where responsiveness is important. To address this, we follow the same approach as in §4.4.2 and consider the case where at most k out of $n = 2k + 1$ processes may fail. Then, by Theorem 5.4, this assumption implies the existence of a connected core in every admissible failure pattern, which we can exploit to design an upper bound with stronger latency guarantees, even in the presence of channel failures. This result is considerably more challenging than the upper bound in §4.4.2, and we prove it in the rest of the chapter.

5.4 Upper Bound under Bounded Process Failures

In this section, we present a consensus protocol in the model $\mathcal{M}_{\text{PEF}}$ for systems in which any minority of processes may fail. That is, we consider a fail-prone system $\mathcal{F}$ such that 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.

As explained in Chapter 1, classical failure detectors and leader election mechanisms are ineffective in the presence of flaky channels. Instead, our protocol builds on a view synchronizer [17, 18, 54, 55], which divides execution into a sequence of views, each led by a designated process. At a high level, the synchronizer aims to gather sufficiently many correct processes with enough connectivity (e.g., those in $\mathcal{S}$) into a view led by a well-connected leader and to keep them in that view long enough to reach agreement.

Supporting this in the presence of flaky channels is nontrivial, since processes outside 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 view $v+1$ if some process from $\mathcal{S}$ has attempted to advance from view 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 view $\mathcal{V}$ and duration 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 5.3: Synchronizer properties satisfied in executions with connected core $\mathcal{S}$.

connected core – such as process b in Figures 1.1b-c – may fail to observe progress by the leader of a functional view and request a premature view change. To address this, we first present the specification (§5.4.1) and implementation (§5.4.2) of a synchronizer that tolerates such disruptions. We then use it to build a consensus protocol (§5.4.4) that operates correctly despite this adversarial failure mode.

5.4.1 Synchronizer Specification

We consider a synchronizer interface defined in [17, 54]. 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 new_view(v) at a process, informing it to enter view v. To trigger these, the synchronizer provides a function advance(), which signals that the process wishes to *advance* to a higher view. We assume that a process does not invoke advance() twice without an intervening new_view notification.

In Figure 5.3 we give a specification of a view synchronizer for the system model $\mathcal{M}_{\text{PEF}}$, adapted from the Byzantine setting [17]. The *Monotonicity* property ensures that a process's view can only increase. *Validity* requires that a process may enter view $v+1$ only if some process in the connected core $\mathcal{S}$ has called advance in v, preventing poorly connected processes from forcing spurious view changes (e.g., process b in Figure 1.1c, where $\mathcal{S} = \{a, c\}$). *Bounded Entry* ensures that if some process in $\mathcal{S}$ enters a view v, then all processes in $\mathcal{S}$ will do so within at most d time units (for a constant d), provided that no process in $\mathcal{S}$ advances further within that interval; otherwise, some processes may skip v entirely. Bounded Entry only holds from some view $\mathcal{V}$ onward, since a synchronizer may not guarantee it before GST. The *Startup* property ensures that if more than $\frac{n}{2}$ processes in $\mathcal{S}$ attempt to advance from view 0, then some process in $\mathcal{S}$ enters view 1. Finally, *Progress* states that if a process in $\mathcal{S}$ enters view v, and more than $\frac{n}{2}$ processes in $\mathcal{S}$ eventually call advance while in v, then some process in $\mathcal{S}$ will enter view $v+1$ (e.g., a and c in Figure 1.1c).

Together, these properties ensure the liveness of consensus despite flaky channels. Informally, Progress lets processes iterate over views until finding one with a well-connected leader; Bounded Entry allows all members of $\mathcal{S}$ to converge quickly into that view; and Validity ensures they remain there despite disruptions from poorly connected processes.

5.4.2 Synchronizer Implementation

In Figure 5.4 we present an algorithm that implements the specification in Figure 5.3 under the model $\mathcal{M}_{\text{PEF}}$. The implementation requires only bounded space, even though correct channels are *eventually* timely and may drop messages before GST.

Each process stores its current view in curr_view and maintains an array views that records, for every process, the highest view it wishes to advance to. When a process invokes advance (line 1), the synchronizer does not immediately switch views. Instead, it updates its entry in views and propagates the entire array in a WISH message (line 3), thereby advertising its intent to advance. Upon receiving a WISH message (line 6), a process merges the received information into its own views array, keeping the maximum view for each entry (line 8). This mechanism ensures that information is relayed even between processes that lack direct connectivity through a correct channel.

Since the membership of $\mathcal{S}$ is unknown to the processes, the synchronizer cannot initiate a view change based on a single advance() call: it cannot tell whether this process belongs to $\mathcal{S}$. Instead, we require wishes to advance from a majority of processes, which guarantees that at least one comes from $\mathcal{S}$. Concretely, upon receiving a WISH message, a process computes v' as the $(\lfloor \frac{n}{2} \rfloor + 1)$-st highest view in views (line 9). Thus, at least one process in $\mathcal{S}$ must wish to advance to a view $\geq v'$. The process then enters v' if $v' >$ curr_view (line 12). Note that a process may be forced to switch views even if it did not invoke advance, which helps lagging processes catch up. To satisfy Bounded Entry, the process disseminates the information that triggered its entry into the new view (line 13), ensuring that others follow promptly. Finally, to cope with message loss before GST, each process periodically resends its views array (line 4).

It remains to show that the synchronizer satisfies Progress. This property requires that more than half of the processes in $\mathcal{S}$ invoke advance. Because these processes are sufficiently well-connected, the corresponding WISH messages eventually propagate within $\mathcal{S}$, enabling

```
1  function advance():
2      views[$i$] $\leftarrow$ curr_view $+ 1$
3      send WISH(views) to all
4  periodically    $\triangleright$ every $\rho$ time units
5      send WISH(views) to all
6  when received WISH($V$)
7      for $p_j \in \mathcal{P}$ do
8          views[$j$] $\leftarrow \max($views[$j$]$, V[j])$
9      $v' \leftarrow \max\{v \mid \exists p_j.$ views[$j$] $= v \wedge |\{p_k \mid$ views[$k$] $\geq v\}| > \frac{n}{2}\}$
10     if $v' >$ curr_view then
11         curr_view $\leftarrow v'$
12         trigger new_view($v'$)
13         send WISH(views) to all
```

Figure 5.4: Synchronizer protocol at a process p_i.

the guard at line 10 (e.g., processes a and c in Figure 1.1c). Note that Progress would not hold if it only required a majority of advance calls from arbitrary processes: for example, if the majority included a and b, we could not guarantee that all corresponding WISH messages propagate.

5.4.3 Synchronizer Correctness

Theorem 5.5. *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 5.4 over the model $\mathcal{M}_{\text{PEF}}$ satisfies the properties in Figure 5.3.*

To prove Theorem 5.5, let $\mathcal{F}$ be a fail-prone system satisfying the stated condition: for each $f \in \mathcal{F}$, the graph $\mathcal{G} \setminus f$ contains a connected core. Fix an arbitrary failure pattern $f \in \mathcal{F}$ and let $\mathcal{S}$ be its corresponding connected core. We show that every f-compliant fair execution of the algorithm in Figure 5.4 over the model $\mathcal{M}_{\text{PEF}}$ satisfies the properties in Figure 5.3. The argument relies on the following propositions, whose proofs are straightforward and thus we omit them.

Proposition 5.6. *If a message WISH(V) is sent at a time t, then $V[i] \leq p_i$.curr_view$(t) + 1$ for every process p_i.*

Proposition 5.7. *If a process p_i has p_i.views$(t)[j] > 0$, then there exists an array V and a time $t' \leq t$ such that $V[j] = p_i$.views$(t)[j]$ and p_j sends WISH(V) at t'.*

Proposition 5.8. *If a process p_i enters a view v, then there exists a process $p_j \in \mathcal{S}$, an array V and a time $t < E_i(v)$ such that $V[j] \geq v$ and p_j sends WISH(V) at t.*

Proposition 5.9. *If a process p_i sends* `WISH`(V) *with $V[i] = v + 1$ at a time t, then $A_i(v){\downarrow} \wedge A_i(v) \le t$.*

Lemma 5.10 (Validity)**.** *A process only enters $v+1$ if some process from $\mathcal{S}$ has attempted to advance from v:*

$$\forall i, v.\ E_i(v+1){\downarrow} \implies A^{\mathcal{S}}_{\text{first}}(v){\downarrow} \wedge A^{\mathcal{S}}_{\text{first}}(v) < E_i(v+1).$$

Proof. Let i and v be such that $E_i(v+1){\downarrow}$. By Proposition 5.8, there exists a process $p_j \in \mathcal{S}$, an array V and a time $t < E_i(v+1)$ such that $V[j] \ge v+1$ and p_j sends `WISH`(V) at t. Suppose $V[j] = v+1$. Then, by Proposition 5.9, $A_j(v){\downarrow} \wedge A_j(v) \le t$. Therefore, $A^{\mathcal{S}}_{\text{first}}(v){\downarrow}$ and

$$A^{\mathcal{S}}_{\text{first}}(v) \le A_j(v) \le t < E_i(v+1).$$

Suppose now $V[j] > v+1$. By Proposition 5.6,

$$p_j.\mathsf{curr\_view}(t) \ge V[j] - 1 > v.$$

Let p_k be the first process to enter a view $v_k > v$ at a time $t_k \le t$. The process p_k enters v_k upon executing line 12, and by lines 9 and 10, $p_k.\mathsf{views}(t_k)$ includes more than $\frac{n}{2}$ entries $\ge v_k$. Because there are at most $\frac{n}{2}$ processes not in $\mathcal{S}$, there exists a process $p_l \in \mathcal{S}$ and a view $v_l \ge v_k$ such that $p_k.\mathsf{views}(t_k)[l] = v_l$. By Proposition 5.7, there exists an array V' and a time $t_l \le t_k$ such that $V'[l] = v_l$ and p_l sends `WISH`(V') at t_l. By Proposition 5.6,

$$p_l.\mathsf{curr\_view}(t_l) \ge v_l - 1 \ge v_k - 1 \ge v.$$

Because p_k is the first process to enter a view $> v$ at t_k and $t_l \le t_k$ then $p_l.\mathsf{curr\_view}(t_l) \le v$. Therefore, $p_l.\mathsf{curr\_view}(t_l) = v$ and $v_l = v+1$. Then, by Proposition 5.9, $A_l(v){\downarrow} \wedge A_l(v) \le t_l$. Therefore, $A^{\mathcal{S}}_{\text{first}}(v){\downarrow}$ and

$$A^{\mathcal{S}}_{\text{first}}(v) \le A_l(v) \le t_l \le t_k \le t < E_i(v+1).$$

□

Lemma 5.11. *$\mathcal{S}$ does not skip views:*

$$\forall v, v'.\ 0 < v < v' \wedge E_{\text{first}}(v'){\downarrow} \implies E^{\mathcal{S}}_{\text{first}}(v){\downarrow} \wedge E^{\mathcal{S}}_{\text{first}}(v) < E_{\text{first}}(v').$$

Proof. Let $v' \ge 2$ and assume that a process enters v', so that $E_{\text{first}}(v'){\downarrow}$. We prove by induction that for each k satisfying $1 \le k \le v'-1$, some process in $\mathcal{S}$ enters $v'-k$ earlier than $E_{\text{first}}(v')$ and thus

$$E^{\mathcal{S}}_{\text{first}}(v'-k){\downarrow} \wedge E^{\mathcal{S}}_{\text{first}}(v'-k) < E_{\text{first}}(v').$$

For the base case assume that a process enters v'. Then by Lemma 5.10, there exists a process $p_i \in \mathcal{S}$ such that

$$A_i(v'-1){\downarrow} \wedge A_i(v'-1) < E_{\text{first}}(v').$$

Because $p_i.\textsf{curr\_view}(A_i(v'-1)) = v'-1$ then

$$E_i(v'-1)\downarrow \;\wedge\; E_i(v'-1) \leq A_i(v'-1) < E_{\text{first}}(v').$$

For the inductive step assume that the required holds for some k so that

$$E^{\mathcal{S}}_{\text{first}}(v'-k)\downarrow \;\wedge\; E^{\mathcal{S}}_{\text{first}}(v'-k) < E_{\text{first}}(v').$$

Then by Lemma 5.10, there exists a process $p_i \in \mathcal{S}$ such that

$$A_i(v'-k-1)\downarrow \;\wedge\; A_i(v'-k-1) < E^{\mathcal{S}}_{\text{first}}(v'-k).$$

Because $p_i.\textsf{curr\_view}(A_i(v'-k-1)) = v'-k-1$ then

$$E_i(v'-k-1)\downarrow \;\wedge\; E_i(v'-k-1) \leq A_i(v'-k-1) < E^{\mathcal{S}}_{\text{first}}(v'-k) < E_{\text{first}}(v').$$

□

Lemma 5.12. *Let* $\Delta = \texttt{diameter}(\mathcal{S})\delta$*. Consider a view* $v > 0$ *and assume that* v *is entered by some process in* $\mathcal{S}$*. If* $E^{\mathcal{S}}_{\text{first}}(v) \geq \textsf{GST}$ *and no process in* $\mathcal{S}$ *attempts to advance from* v *before* $E^{\mathcal{S}}_{\text{first}}(v) + \Delta$*, then all processes in* $\mathcal{S}$ *enter* v *and* $E^{\mathcal{S}}_{\text{last}}(v) \leq E^{\mathcal{S}}_{\text{first}}(v) + \Delta$*.*

Proof. Suppose there exists a process p_i and a time $t \leq E^{\mathcal{S}}_{\text{first}}(v)+\Delta$ such that $p_i.\textsf{curr\_view}(t) = v' > v$. By Lemma 5.11,

$$E_{\text{first}}(v+1)\downarrow \;\wedge\; E_{\text{first}}(v+1) \leq E_{\text{first}}(v') \leq t.$$

Thus, by Lemma 5.10,

$$A^{\mathcal{S}}_{\text{first}}(v)\downarrow \;\wedge\; A^{\mathcal{S}}_{\text{first}}(v) < E_{\text{first}}(v+1) \leq t \leq E^{\mathcal{S}}_{\text{first}}(v) + \Delta,$$

contradicting the assumption that no process in $\mathcal{S}$ attempts to advance from v before $E^{\mathcal{S}}_{\text{first}}(v) + \Delta$. Therefore, for all times $t \leq E^{\mathcal{S}}_{\text{first}}(v) + \Delta$ and processes p_i, $p_i.\textsf{curr\_view}(t) \leq v$.

Let p_i be the process in $\mathcal{S}$ to enter v at the time $E^{\mathcal{S}}_{\text{first}}(v)$ and $p_j \in \mathcal{S}$. We prove by induction on the distance d from p_i to p_j (considering the reliable links that make up $\mathcal{S}$) that

$$E_j(v)\downarrow \;\wedge\; E_j(v) \leq E_i(v) + d\delta.$$

Since p_i is the only process at distance 0 from p_i, the result trivially holds for $d = 0$. Assume now the result holds for all processes in $\mathcal{S}$ at a distance d from p_i and suppose the distance from p_i to p_j is $d+1$. Therefore, there exists a process $p_k \in \mathcal{S}$ such that the distance from p_i to p_k is d and the distance from p_k to p_j is 1. By induction hypothesis p_k enters v no later than $E_i(v) + d\delta$. The process p_k enters view v upon executing line 12, and by lines 9 and 10, $p_k.\textsf{views}(E_k(v))$ includes more than $\frac{n}{2}$ entries $\geq v$.

Line 13 guarantees that upon entering v, p_k sends a $\texttt{WISH}(p_k.\textsf{views}(E_k(v)))$ to every process. Since the link between p_k and p_j is timely after $\textsf{GST}$, the $\texttt{WISH}$ message is received by p_j at a time $t_j \leq E_i(v) + (d+1)\delta$. Upon receipt of the $\texttt{WISH}$ message, p_j executes lines 7 and 8, ensuring that $p_j.\textsf{views}(t_j) \geq p_k.\textsf{views}(E_k(v))$. Therefore, $p_j.\textsf{views}(t_j)$ includes more than

$\frac{n}{2}$ entries $\geq v$, and thus p_j is guaranteed to enter a view $v' \geq v$ no later than t_j. Because $t_j \leq E^{\mathcal{S}}_{\text{first}}(v) + \Delta$ and no process can have a view $> v$ before $E^{\mathcal{S}}_{\text{first}}(v) + \Delta$, it follows that $v' = v$. Therefore, p_j enters v no later than t_j and thus

$$E_j(v) \leq t_j \leq E_i(v) + (d+1)\delta.$$

Thus, all processes in $\mathcal{S}$ enter v and $E^{\mathcal{S}}_{\text{last}}(v) \leq E^{\mathcal{S}}_{\text{first}}(v) + \Delta$. □

Lemma 5.13 (Bounded Entry)**.** *For some $\mathcal{V}$ and d, if a process from $\mathcal{S}$ enters a view $v \geq \mathcal{V}$ and no process from $\mathcal{S}$ attempts to advance to a higher view within time d, then every process from $\mathcal{S}$ will enter v within d:*

$$\exists \mathcal{V}, d.\ \forall v \geq \mathcal{V}.\ E^{\mathcal{S}}_{\text{first}}(v){\downarrow} \ \wedge \neg(A^{\mathcal{S}}_{\text{first}}(v) < E^{\mathcal{S}}_{\text{first}}(v) + d) \implies$$
$$(\forall p_i \in \mathcal{S}.\ E_i(v){\downarrow}) \wedge (E^{\mathcal{S}}_{\text{last}}(v) \leq E^{\mathcal{S}}_{\text{first}}(v) + d).$$

Proof. Let

$$\mathcal{V} = \max\{v \mid E^{\mathcal{S}}_{\text{first}}(v){\downarrow} \ \wedge E^{\mathcal{S}}_{\text{first}}(v) < \mathsf{GST}\} + 1$$

so that

$$\forall v \geq \mathcal{V}.\ E^{\mathcal{S}}_{\text{first}}(v){\downarrow} \implies E^{\mathcal{S}}_{\text{first}}(v) \geq \mathsf{GST}$$

and $d = \texttt{diameter}(\mathcal{S})\delta$. Then, by Lemma 5.12, Bounded Entry holds. □

Given arrays V and V' we say that $V \leq V'$ if and only if $\forall i.\ V[i] \leq V'[i]$.

Lemma 5.14. *If a process sends $\texttt{WISH}(V)$ before sending $\texttt{WISH}(V')$, then $V \leq V'$.*

Proof. Let $p_j \in \mathcal{P}$ and t and t' be the times at which a process p_i sends $\texttt{WISH}(V)$ and $\texttt{WISH}(V')$, respectively. Notice that $V[j] = p_i.\mathsf{views}(t)[j]$ and $V'[j] = p_i.\mathsf{views}(t')[j]$. Since $p_i.\mathsf{views}[j]$ is non-decreasing, as guaranteed by lines 2 and 8,

$$p_i.\mathsf{view}(t)[j] \leq p_i.\mathsf{view}(t')[j].$$

Therefore, $V[j] \leq V'[j]$. Since p_j was picked arbitrarily, $V \leq V'$. □

Lemma 5.15. *For all processes $p_i \in \mathcal{S}$, times $t \geq \mathsf{GST} + \rho$ and arrays V, if p_i sends $\texttt{WISH}(V)$ at a time $\leq t$, then there exists an array $V' \geq V$ and a time t' such that $\mathsf{GST} \leq t' \leq t$ and p_i sends $\texttt{WISH}(V')$ at t'.*

Proof. Let $s \leq t$ be the time at which p_i sends $\texttt{WISH}(V)$. If $s \geq \mathsf{GST}$, then choosing $t' = s$ and $V' = V$ validates the lemma. Assume now that $s < \mathsf{GST}$. Since after GST the p_i's local clock advances at the same rate as real time, there exists a time t' satisfying $\mathsf{GST} \leq t' \leq t$ such that p_i executes the retransmission code in lines 4 and 5 at t'. Then, there exists an array V' such that p_i sends $\texttt{WISH}(V')$ at t'. By Lemma 5.14, $V' \geq V$. □

Lemma 5.16 (Startup)**.** *If more than $\frac{n}{2}$ processes from $\mathcal{S}$ invoke* $\mathsf{advance}$*, then some process from $\mathcal{S}$ will enter view 1:*

$$(\exists P \subseteq \mathcal{S}.\ |P| > \frac{n}{2} \ \wedge \ (\forall p_i \in P.\ A_i(0){\downarrow})) \implies E^{\mathcal{S}}_{\text{first}}(1){\downarrow}.$$

Proof. Assume by contradiction that there exists a set $P \subseteq \mathcal{S}$ of more than $\frac{n}{2}$ processes such that $\forall p_i \in P.\ A_i(0)\downarrow$, and no process in $\mathcal{S}$ enters view 1. By Lemma 5.11, the latter implies

$$\forall v > 0.\ \forall p_i \in \mathcal{S}.\ E_i(v)\uparrow . \tag{5.1}$$

We now show that for every $p_i \in \mathcal{S}$ and $p_j \in P$, there exists an array V_j and a time t_j such that $V_j[j] \geq 1$ and p_i receives $\mathtt{WISH}(V_j)$ at t_j. Fix $p_j \in P$. We prove the result by induction on the distance d from p_j to p_i (considering the reliable links that make up $\mathcal{S}$).

Since $A_j(0)\downarrow$, lines 2 and 3 guarantee that there exists an array V_j and a time t_j such that $V_j[j] = 1$ and p_j sends $\mathtt{WISH}(V_j)$ at t_j. Since messages sent to itself are always delivered and p_j is the only process at distance 0 from p_j, the result trivially holds for $d = 0$.

Assume now the result holds for all processes at a distance d from p_j and suppose that $p_i \in \mathcal{S}$ is a process at a distance $d+1$ from p_j. Therefore, there exists a process $p_k \in \mathcal{S}$ such that the distance from p_j to p_k is d and the distance from p_k to p_i is 1. By induction hypothesis there exists an array V_j and a time t_j such that $V_j[j] \geq 1$ and p_k receives $\mathtt{WISH}(V_j)$ at t_j. Upon receipt of the $\mathtt{WISH}$ message, p_k executes lines 7 and 8, ensuring that

$$p_k.\mathsf{views}(t_j)[j] \geq V[j] \geq 1.$$

Because the periodic handler in line 4 fires every ρ units of time, there exists an array $V_{k'}$ and a time $t_{k'} \leq t_j + \rho$ such that $V_{k'}[j] \geq 1$ and p_k sends $\mathtt{WISH}(V_{k'})$ at $t_{k'}$. Let $T_k = \max(\mathsf{GST} + \rho, t'_k)$. By Lemma 5.15, there exists an array $V_k \geq V_{k'}$ and a time t_k such that $\mathsf{GST} \leq t_k \leq T_k$ and p_k sends $\mathtt{WISH}(V_k)$ at t_k. Since the link between p_k and p_i is timely after GST, the $\mathtt{WISH}(V_k)$ is received by p_i no later than $t_k + \delta$. And because $V_k \geq V_{k'}$ then

$$V_k[j] \geq V_{k'}[j] \geq 1.$$

This completes the induction.

Let $p_l \in \mathcal{S}$ be any process and, for each $p_i \in P$, let V_i and t_i be such that $V_i[i] \geq 1$ and p_l receives $\mathtt{WISH}(V_i)$ at t_i. Let $T = \max t_i$. Lines 7 and 8 guarantee that $p_l.\mathsf{views}(T)[i] \geq 1$ for each $p_i \in P$. Since $|P| > \frac{n}{2}$, $p_k.\mathsf{views}(T)$ includes more than $\frac{n}{2}$ entries ≥ 1, and thus, $p_l.v'(T) \geq 1$. By (5.1), $p_l.\mathsf{curr\_view}(T) = 0$. Hence, line 10 ensures that p_l enters a view ≥ 1 by T, contradicting (5.1). ☐

Lemma 5.17 (Progress). *If a process from $\mathcal{S}$ enters a view v and, for some set $P \subseteq \mathcal{S}$ of more than $\frac{n}{2}$ processes, any process in P that enters v eventually invokes* $\mathsf{advance}$*, then some process from $\mathcal{S}$ will enter $v + 1$:*

$$\forall v.\ E^{\mathcal{S}}_{\text{first}}(v)\downarrow \wedge (\exists P \subseteq \mathcal{S}.\ |P| > \frac{n}{2} \wedge (\forall p_i \in P. E_i(v)\downarrow \implies A_i(v)\downarrow)) \implies E^{\mathcal{S}}_{\text{first}}(v+1)\downarrow .$$

Proof. Assume by contradiction that there exists a set $P \subseteq \mathcal{S}$ of more than $\frac{n}{2}$ processes such that

$$\forall p_i \in P.\ E_i(v)\downarrow \implies A_i(v)\downarrow \wedge\ E^{\mathcal{S}}_{\text{first}}(v)\downarrow ,$$

and no process in $\mathcal{S}$ enters view $v + 1$. By Lemma 5.11, the latter implies

$$\forall v' > v.\ \forall p_i \in \mathcal{S}.\ E_i(v')\uparrow . \tag{5.2}$$

Let $p_i \in \mathcal{S}$ be the first process to enter v. We show that every process $p_j \in \mathcal{S}$ enters v by induction on the distance d from p_i to p_j (considering the reliable links that make up $\mathcal{S}$). Since p_i is the only process at distance 0 from p_i, the result trivially holds for $d = 0$. Assume now the result holds for all processes in $\mathcal{S}$ at a distance d from p_i and suppose the distance from p_i to p_j is $d + 1$. Therefore, there exists a process $p_k \in \mathcal{S}$ such that the distance from p_i to p_k is d and the distance from p_k to p_j is 1. By induction hypothesis p_k enters v. The process p_k enters view v upon executing line 12, and by lines 9 and 10, $p_k.\mathsf{views}(E_k(v))$ includes more than $\frac{n}{2}$ entries $\geq v$.

Line 13 guarantees that upon entering v, p_k sends $\texttt{WISH}(p_k.\mathsf{views}(E_k(v)))$ to every process. Let $T_k = \max(\mathsf{GST} + \rho, E_k(v))$. By Lemma 5.15, there exists an array $V_k \geq p_k.\mathsf{views}(E_k(v))$ and a time t_k such that $\mathsf{GST} \leq t_k \leq T_k$ and p_k sends $\texttt{WISH}(V_k)$ at t_k. Since the link between p_k and p_j is timely after GST, the $\texttt{WISH}$ message is received by p_j at a time t_j. Upon receipt of the $\texttt{WISH}$ message, p_j executes lines 7 and 8, ensuring that

$$p_j.\mathsf{views}(t_j) \geq V_k \geq p_k.\mathsf{views}(E_k(v)).$$

Therefore, $p_j.\mathsf{views}(t_j)$ includes more than $\frac{n}{2}$ entries $\geq v$, and thus p_j is guaranteed to enter a view $v' \geq v$ no later than t_j. By (5.2), p_j never enters a view $> v$ and thus p_j enters v. This completes the induction. Because every process from $\mathcal{S}$ enters v, we have

$$\forall p_i \in P.\ E_i(v)\!\downarrow$$

and thus

$$\forall p_i \in P.\ A_i(v)\!\downarrow.$$

We now show that for every $p_i \in \mathcal{S}$ and $p_j \in P$, there exists an array V_j and a time t_j such that $V_j[j] \geq v+1$ and p_i receives $\texttt{WISH}(V_j)$ at t_j. Fix $p_j \in P$. We prove the result by induction on the distance d from p_j to p_i (considering the reliable links that make up $\mathcal{S}$).

Since $A_j(v)\!\downarrow$, lines 2 and 3 guarantee that there exists an array V_j and a time t_j such that $V_j[j] = v + 1$ and p_j sends $\texttt{WISH}(V_j)$ at t_j. Since messages sent to itself are always delivered and p_j is the only process at distance 0 from p_j, the result trivially holds for $d = 0$.

Assume now the result holds for all processes at a distance d from p_j and suppose that $p_i \in \mathcal{S}$ is a process at a distance $d + 1$ from p_j. Therefore, there exists a process $p_k \in \mathcal{S}$ such that the distance from p_j to p_k is d and the distance from p_k to p_i is 1. By induction hypothesis there exists an array V_j and a time t_j such that $V_j[j] \geq v + 1$ and p_k receives $\texttt{WISH}(V_j)$ at t_j. Upon receipt of the $\texttt{WISH}$ message, p_k executes lines 7 and 8, ensuring that

$$p_k.\mathsf{views}(t_j)[j] \geq V[j] \geq v + 1.$$

Because the periodic handler in line 4 fires every ρ units of time, there exists an array $V_{k'}$ and a time $t_{k'} \leq t_j + \rho$ such that $V_{k'}[j] \geq v + 1$ and p_k sends $\texttt{WISH}(V_{k'})$ at $t_{k'}$. Let $T_k = \max(\mathsf{GST} + \rho, t'_k)$. By Lemma 5.15, there exists an array $V_k \geq V_{k'}$ and a time t_k such that $\mathsf{GST} \leq t_k \leq T_k$ and p_k sends $\texttt{WISH}(V_k)$ at t_k. Since the link between p_k and p_i is timely after GST, the $\texttt{WISH}(V_k)$ is received by p_i no later than $t_k + \delta$. And because $V_k \geq V_{k'}$ then

$$V_k[j] \geq V_{k'}[j] \geq v + 1.$$

```
1  on startup
2      advance()
3  function propose($x$):
4      my_val ← $x$
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, aview, val)
11     phase ← ENTERED
12 when the timer decision_timer expires
13     timeout ← timeout + $\gamma$
14     advance()
15 periodically        ▷ every $\rho$ time units
16     send STATE(M1B, M2A, M2B) to all
17 when received STATE($V1B$, $V2A$, $V2B$)
18     for $p_j \in \mathcal{P}$ 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$[j]$ ← $V2B[j]$
```

Figure 5.5: Consensus protocol at a process p_i (part 1).

This completes the induction.

Let $p_l \in \mathcal{S}$ be any process and, for each $p_i \in P$, let V_i and t_i be such that $V_i[i] \geq v+1$ and p_l receives WISH(V_i) at t_i. Let $T = \max t_i$. Lines 7 and 8 guarantee that p_l.views$(T)[i] \geq v+1$ for each $p_i \in P$. Since $|P| > \frac{n}{2}$, p_k.views(T) includes more than $\frac{n}{2}$ entries $\geq v+1$, and thus, $p_l.v'(T) \geq v+1$. By (5.2), p_l.curr_view$(T) \leq v$. Hence, line 10 ensures that p_l enters a view $\geq v+1$ by T, contradicting (5.2). □

Proof of Theorem 5.5. The guard at line 10 ensures Monotonicity. Validity, Bounded Entry, Startup and Progress are given by Lemmas 5.10, 5.13, 5.16 and 5.17, respectively. □

5.4.4 Consensus Protocol

In Figures 5.5–5.6 we present an implementation of consensus over the model $\mathcal{M}_{\text{PEF}}$, resilient to the failure of any minority of processes. The protocol is a variation of single-decree Paxos [47], where liveness is ensured with the help of a view synchronizer. The protocol

```
25 when $\mathsf{phase} = \mathtt{ENTERED} \wedge \mathsf{leader}(\mathsf{view}) = p_i \wedge |\{p_j \mid p_j \in \mathcal{P} \wedge \mathsf{M1B}[j].view = \mathsf{view}\}| > \frac{n}{2}$
26     $Q \leftarrow \{p_j \mid p_j \in \mathcal{P} \wedge \mathsf{M1B}[j].view = \mathsf{view}\}$
27     if $\forall p_j.\ p_j \in Q \implies \mathsf{M1B}[j].val = \bot$ then
28         if $\mathsf{my\_val} = \bot$ then return
29         $\mathsf{M2A}[i] \leftarrow (\mathsf{view}, \mathsf{my\_val})$
30     else
31         let $p_j \in Q$ be such that
             $\mathsf{M1B}[j].val \neq \bot \wedge \forall p_k \in Q.\ \mathsf{M1B}[k].cview \leq \mathsf{M1B}[j].cview$
32         $\mathsf{M2A}[i] \leftarrow (\mathsf{view}, \mathsf{M1B}[j].val)$
33     $\mathsf{phase} \leftarrow \mathtt{PROPOSED}$

34 when $\mathsf{phase} \in \{\mathtt{ENTERED}, \mathtt{PROPOSED}\} \wedge p_l = \mathsf{leader}(\mathsf{view}) \wedge \mathsf{M2A}[l].view = \mathsf{view}$
35     $(\mathsf{aview}, \mathsf{val}) \leftarrow \mathsf{M2A}[l]$
36     $\mathsf{M2B}[i] \leftarrow (\mathsf{aview}, \mathsf{val})$
37     $\mathsf{phase} \leftarrow \mathtt{ACCEPTED}$

38 when $\exists v, x.\ v \geq \mathsf{view} \wedge |\{p_j \mid p_j \in \mathcal{P} \wedge \mathsf{M2B}[j] = (v, x)\}| > \frac{n}{2}$
39     $\mathsf{val} \leftarrow x$
40     $\mathsf{stop\_timer}(\mathsf{decision\_timer})$
41     $\mathsf{phase} \leftarrow \mathtt{DECIDED}$
```

Figure 5.6: Consensus protocol at a process p_i (part 2).

operates in a succession of views produced by the synchronizer. Each view v has a fixed leader $\mathsf{leader}(v) = p_{((v-1) \bmod n)+1}$, responsible for proposing a value to the other processes, which then vote on the proposal. Processes monitor the leader's behavior and request the synchronizer to advance to another view if they suspect that the leader is faulty or poorly connected. As in §4.4.2, a notable aspect of this implementation is that, despite relying on gossip, it uses only bounded space and its progress depends directly on message delivery delays and the rate of state exchange.

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

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, since leader election is controlled by the synchronizer. Because correct processes may not be directly connected by correct channels, each process must forward information received from others. Moreover, since correct channels are only *eventually* timely, this forwarding must be repeated periodically. If implemented naively, this would require unbounded space to store all messages pending forwarding. Instead, it suffices to store, for each message type and sender, only the message with the highest view. These are kept in the arrays M1B, M2A, and M2B.

For clarity, the pseudocode in Figures 5.5–5.6 separates computation from communication.

Most handlers do not send messages but only update the arrays M1B, M2A, and M2B. Then, instead of sending individual 1B, 2A, or 2B messages as in Paxos, a process periodically sends the whole arrays in one STATE message (line 16). Upon receipt of a STATE message (line 17), a process merges the information into its arrays, keeping the entries with the highest view. This ensures information is propagated between processes without direct connectivity, while still using bounded space.

Normal protocol operation. When the synchronizer instructs 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 selects the one 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_val (line 29); if not, 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 aview, and notifies all processes 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 a decision has been reached and sets phase = DECIDED. If there is an ongoing propose() invocation, the condition at line 5 is 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 poorly connected. To this end, when a process enters a view, it sets a timer decision_timer of duration timeout (line 9) and stops it once a decision is reached (line 40). If the timer expires before a decision is made, the process invokes advance (line 14). A process may incorrectly suspect a correct leader if timeout is initially too short relative to the actual message delay δ, which is unknown to the process. To overcome this, the process increases timeout each time the timer expires (line 13), thereby adapting its suspicion threshold to the network latency.

5.4.5 Correctness

It is easy to see that the protocol in Figures 5.5-5.6 satisfies the safety properties of consensus, as the argument is virtually identical to that of Paxos [47]. We therefore focus on proving liveness.

Theorem 5.18. *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 for each $f \in \mathcal{F}$, $\tau(f) = \mathcal{S}_f$. Then the algorithm in Figures 5.5-5.6 is $(\mathcal{F}, \tau)$-wait-free.*

Fix a failure pattern f and let $\mathcal{S}$ be the corresponding connected core guaranteed to exist by the assumptions of Theorem 5.18. We prove liveness by showing that the protocol establishes properties reminiscent to those of failure detectors [24]. 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 $\mathsf{view} = v$.

Lemma 5.19. *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* $\mathsf{advance}$ *in v.*

Proof. Upon entering v the process p_i starts $\mathsf{decision\_timer}$ (line 9). The process p_i cannot stop the timer at line 9, since together with Monotonicity, this would imply that p_i enters a view higher than v. The process p_i cannot stop the timer at line 40 either, since this would imply that p_i decides in v. Therefore the timer must expire, after which p_i invokes $\mathsf{advance}$ in v (line 14). □

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 never attempt to advance from it.

For a failure pattern f with an associated connected core $\mathcal{S}$, let $\mathsf{diameter}(\mathcal{S})$ 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, and $\Delta = (\delta + \rho)\mathsf{diameter}(\mathcal{S})$. Finally, let $\mathsf{timeout}_i(v)$ denote the value of $\mathsf{timeout}$ at the process p_i while in view v.

Lemma 5.20. *Let $v \geq \mathcal{V}$ be a view such that $\mathsf{leader}(v) \in \mathcal{S}$, $E^{\mathcal{S}}_{\mathsf{first}}(v) \geq \mathsf{GST}$ and $\mathsf{leader}(v)$ invokes* $\mathsf{propose}$ *no later than $E^{\mathcal{S}}_{\mathsf{first}}(v)$. If at each process $p_i \in \mathcal{S}$ that enters v we have $\mathsf{timeout}_i(v) > d + 3\Delta$, then no process in $\mathcal{S}$ invokes* $\mathsf{advance}$ *in v.*

Lemma 5.20 together with Validity implies that all processes in $\mathcal{S}$ will stay in a view where progress is possible provided the timers are high enough. However, to apply it we need to argue that, if processes from $\mathcal{S}$ keep changing views due to lack of progress, all of them will increase their timeouts high enough to satisfy the bounds in Lemma 5.20. To this end, we prove the following generalization of Lemma 5.20: in a sufficiently high view v with a leader from the connected core, if the timeout at a process from $\mathcal{S}$ that enters v is high enough, then this process cannot be the first to initiate a view change. Hence, for the protocol to enter another view, some other process with a lower timeout must call $\mathsf{advance}$ and thus increase its duration.

Lemma 5.21. *Let $v \geq \mathcal{V}$ be a view such that $\mathsf{leader}(v) \in \mathcal{S}$, $E^{\mathcal{S}}_{\mathsf{first}}(v) \geq \mathsf{GST}$ and $\mathsf{leader}(v)$ invokes* $\mathsf{propose}$ *no later than $E^{\mathcal{S}}_{\mathsf{first}}(v)$. If $p_i \in \mathcal{S}$ enters v and $\mathsf{timeout}_i(v) > d + 3\Delta$, then p_i is not the first process in $\mathcal{S}$ to invoke* $\mathsf{advance}$ *in v.*

The proof of Lemma 5.21 relies on the following technical proposition. It states that the processes from $\mathcal{S}$ exchange their most up to date information within time Δ, as long as they do not enter a higher view.

Proposition 5.22. *Let $t \geq \mathsf{GST}$ be such that no process in $\mathcal{S}$ enters a view higher than v before $t + \Delta$. Then the following hold:*

1. $\forall p_i, p_j, p_k \in \mathcal{S}.\quad p_i.\mathsf{M1B}(t)[k].\mathit{view} = v \Longrightarrow p_j.\mathsf{M1B}(t+\Delta)[k] = p_i.\mathsf{M1B}(t)[k]$.
2. $\forall p_i, p_j \in \mathcal{S}.\ \forall p_k.\ p_i.\mathsf{M2A}(t)[k].\mathit{view} = v \Longrightarrow p_j.\mathsf{M2A}(t+\Delta)[k] = p_i.\mathsf{M2A}(t)[k]$.
3. $\forall p_i, p_j \in \mathcal{S}.\ \forall p_k.\ p_i.\mathsf{M2B}(t)[k].\mathit{view} = v \Longrightarrow p_j.\mathsf{M2B}(t+\Delta)[k] = p_i.\mathsf{M2B}(t)[k]$.

Informally, each process propagates its state every ρ units of time (line 16). Upon receipt of the state (line 17), processes update their vectors with the most up to date information (lines 20, 22 and 24). Because no process from $\mathcal{S}$ is further than $\mathsf{diameter}(\mathcal{S})$ from p_i, its state is guaranteed to be received by every other process in $\mathcal{S}$ no later than $t + \Delta$. And because no process from $\mathcal{S}$ enters a view higher than v at least until after $t + \Delta$, these values remain unchanged until then.

Proof of Proposition 5.22. Let's consider the case of M2B; the other cases are similar. Fix $p_i \in \mathcal{S}$ and let $t \geq \mathsf{GST}$ be a time, $p_k \in \mathcal{P}$ and x be a value such that $p_i.\mathsf{M2B}(t)[k] = (v, x)$. Let $p_j \in \mathcal{S}$ and $\mathsf{dist}(p_i, p_j)$ be the distance from p_i to p_j in $\mathcal{G} \setminus f$. We now show that

$$p_j.\mathsf{M2B}(t + (\rho + \delta)\mathsf{dist}(p_i, p_j))[k] = (v, x).$$

We prove the result by induction on the distance d from p_i to p_j. Since p_i is the only process at distance 0 from p_i, the result trivially holds for $d = 0$. Assume now that the result holds for all processes in $\mathcal{S}$ at distance d from p_i and suppose that the distance from p_i to p_j is $d + 1$. Therefore, there exists a process $p_m \in \mathcal{S}$ such that the distance from p_i to p_m is d and the distance from p_m to p_j is 1. By induction hypothesis,

$$p_m.\mathsf{M2B}(t + (\rho + \delta)d)[k] = (v, x).$$

Because no process in $\mathcal{S}$ enters a view higher than v before $t + \Delta$, the value of

$$p_m.\mathsf{M2B}(t + (\rho + \delta)d)[k]$$

remains unchanged from $t + (\rho + \delta)d$ until $t + \Delta$.

Because the handler at line 15 runs every ρ units of time, there exists a time t_s and a vector *V2B* such that

$$t + (\rho + \delta)d \leq t_s \leq t + (\rho + \delta)d + \rho,$$

$\mathit{V2B}[k] = (v, x)$ and p_m sends `STATE(_, _,` *V2B*`)` at t_s (line 16). Since the channel between p_m and p_j is reliable after GST, the `STATE(_, _,` *V2B*`)` message is received by p_j at a time t_r such that

$$t_r \leq t_s + \delta \leq t + (\rho + \delta)d + \rho + \delta = t + (\rho + \delta)(d + 1).$$

Thus the process p_j executes the handler at line 17 by t_r.

If $p_j.\mathsf{M2B}(t_r)[k].\mathit{view} = v$, and since at most one value can be proposed per view, then $p_j.\mathsf{M2B}(t_r)[k] = (v, x)$. If $p_j.\mathsf{M2B}(t_r)[k].\mathit{view} < v$, then p_j sets $p_j.\mathsf{M2B}(t_r)[k] = (v, x)$ (line 24).

Because no process in $\mathcal{S}$ enters a view higher than v before $t+\Delta$, then $p_j.\mathsf{M2B}(t_r)[k].\mathit{view}$ cannot be greater than v, and also $p_j.\mathsf{M2B}$ remains unchanged from t_r until $t+\Delta$. Therefore,

$$p_j.\mathsf{M2B}(t+(\rho+\delta)(d+1))[k]=(v,x).$$

□

Proof of Lemma 5.21. Let $T=E^{\mathcal{S}}_{\text{first}}(v)+d+3\Delta$. By contradiction, assume that p_i is the first process in $\mathcal{S}$ to call $\mathsf{advance}$ in v. This only occurs if $\mathsf{decision\_timer}$ expires at p_i (line 14). Because p_i is the first process to call $\mathsf{advance}$ in v and $\mathsf{timeout}_i(v)>d+3\Delta$, no process in $\mathcal{S}$ invokes $\mathsf{advance}$ in v until after T. Then by Bounded Entry all processes in $\mathcal{S}$ enter v by

$$E^{\mathcal{S}}_{\text{last}}(v)\le E^{\mathcal{S}}_{\text{first}}(v)+d.$$

By Validity no process can enter $v+1$ until after T, and by Lemma 5.11, the same holds for any view $>v$. Thus, all processes in $\mathcal{S}$ stay in v at least until T. We now show that during this time processes in $\mathcal{S}$ exchange the messages needed for p_i to decide and stop the timer, thereby reaching a contradiction.

Let $p_l=\mathsf{leader}(v)$. We first prove that p_l proposes a value in view v. Indeed, pick an arbitrary process $p_j\in\mathcal{S}$. Upon entering v, p_j sets $\mathsf{M1B}[j]=(v,\_,\_)$ and $\mathsf{phase}=\mathtt{ENTERED}$ (lines 10 and 11). By Proposition 5.22,

$$p_l.\mathsf{M1B}(E_j(v)+\Delta)[j]=p_j.\mathsf{M1B}(E_j(v))[j].$$

Because no process enters a view $>v$ until after T, the value of $p_l.\mathsf{M1B}[j]$ remains unchanged from $E_j(v)+\Delta$ until T. Then, since p_j was picked arbitrarily, by $E^{\mathcal{S}}_{\text{last}}(v)+\Delta$ the array $p_l.\mathsf{M1B}$ contains $|\mathcal{S}|>\frac{n}{2}$ entries with view v. Thus, the precondition at line 25 is satisfied at p_l at a time t_{pl} such that

$$E^{\mathcal{S}}_{\text{first}}(v)\le t_{pl}\le E^{\mathcal{S}}_{\text{last}}(v)+\Delta\le E^{\mathcal{S}}_{\text{first}}(v)+d+\Delta.$$

Since p_l invokes $\mathsf{propose}$ no later than $E^{\mathcal{S}}_{\text{first}}(v)$, $p_l.\mathsf{my\_val}(t_{pl})\neq\bot$. Therefore, by t_{pl} the process p_l sets $\mathsf{M2A}[l]=(v,x)$ for some x, and $\mathsf{phase}=\mathtt{PROPOSED}$ (lines 29, 32 and 33). Moreover, the guard at line 25 and line 33 guarantee that the proposed value x is unique for view v.

We next prove that all processes in $\mathcal{S}$ accept the proposal x made by p_l in view v. Indeed, pick an arbitrary process $p_j\in\mathcal{S}$. By Proposition 5.22,

$$p_j.\mathsf{M2A}(t_{pl}+\Delta)[l]=p_l.\mathsf{M2A}(t_{pl})[l].$$

Thus, the precondition at line 34 is satisfied at p_j at a time t_{aj} such that

$$t_{aj}\le t_{pl}+\Delta\le E^{\mathcal{S}}_{\text{first}}(v)+d+2\Delta.$$

Therefore, by t_{aj} the process p_j sets $\mathsf{M2B}[j]=(v,x)$ and $\mathsf{phase}=\mathtt{ACCEPTED}$ (lines 36 and 37). Finally, we prove that p_i decides in view v. Indeed, pick an arbitrary process $p_j\in\mathcal{S}$. By Proposition 5.22,

$$p_i.\mathsf{M2B}(t_{aj}+\Delta)[j]=p_j.\mathsf{M2B}(t_{aj})[j]=(v,x).$$

Because no process enters a view $> v$ until after T, the value of $p_i.\mathsf{M2B}[j]$ remains unchanged from $t_{aj} + \Delta$ until T. Then, since p_j was picked arbitrarily, by $\max\{t_{aj} \mid p_j \in \mathcal{S}\} + \Delta$ the array $p_i.\mathsf{M2B}$ contains $|\mathcal{S}| > \frac{n}{2}$ entries equal to (v, x). Thus, the precondition at line 38 is satisfied at p_i at a time t_{di} such that

$$t_{di} \le E^{\mathcal{S}}_{\text{first}}(v) + d + 3\Delta = T.$$

Therefore, by t_{di} the process p_i stops the timer $\mathsf{decision\_timer}$ (line 40) before it expires, which contradicts the fact that $\mathsf{decision\_timer}$ expires at p_i while in v. □

Finally, we prove the liveness of the algorithm in Figures 5.5-5.6.

Proof of Theorem 5.18. 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 Figures 5.5-5.6 such that p_j invokes $\mathsf{propose}$ at a time t_{pj} but that the operation never returns. We first prove that in this case the protocol keeps moving through views forever.

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

Proof. Since all processes in $\mathcal{S}$ invoke $\mathsf{advance}$ upon starting, by Startup some process in $\mathcal{S}$ enters view 1. We now show that infinitely many views are entered by some process in $\mathcal{S}$. Assume the contrary, so that there exists a maximal view v entered by any process in $\mathcal{S}$. Let $P \subseteq \mathcal{S}$ be any set of more than $\frac{n}{2}$ processes and consider an arbitrary process $p_i \in P$ that enters v.

Suppose p_i decides in v. Then there exists a time t, a view $v' \ge v$ and a value x such that $p_i.\mathsf{M2B}(t)$ contains more than $\frac{n}{2}$ entries equal to (v', x). Let Q be the set of processes these entries correspond to. Then one of them must be from $\mathcal{S}$. Because no process from $\mathcal{S}$ enters a view $> v$, the precondition at line 34 ensures that $v' = v$. For similar reasons, the value of $p_i.\mathsf{M2B}$ remains unchanged from t onwards. Then there exists a time $t' \ge \mathsf{GST}$ such that $p_i.\mathsf{M2B}(t')$ contains more than $\frac{n}{2}$ entries equal to (v, x).

By Proposition 5.22,

$$p_j.\mathsf{M2B}(t' + \Delta)[k] = p_i.\mathsf{M2B}(t')[k]$$

for each $p_k \in Q$. Because no process from $\mathcal{S}$ enters a view $> v$, the value of $p_j.\mathsf{M2B}$ remains unchanged from $t' + \Delta$ onwards. Then the precondition at line 38 is satisfied at p_j at a time t_{dj}. Therefore, by t_{dj} the process p_j sets $\mathsf{val} = x$ and $\mathsf{phase} = \mathtt{DECIDED}$. But then the $\mathsf{propose}$ operation invoked by p_j returns by $\max\{t_{pj}, t_{dj}\}$, contradicting the assumption that it never does so.

Therefore, p_i never decides in v. Then by Lemma 5.19, p_i eventually invokes $\mathsf{advance}$ in v. Since p_i was picked arbitrarily, we have

$$\forall p_i \in P.\ E_i(v){\downarrow} \implies A_i(v){\downarrow}.$$

By Progress, $E^{\mathcal{S}}_{\text{first}}(v+1){\downarrow}$, which contradicts the fact that v is the maximal view entered by any process in $\mathcal{S}$. Thus, processes in $\mathcal{S}$ keep entering views forever. The claim then follows from Lemma 5.11 ensuring that, if a view is entered by a process in $\mathcal{S}$, then so are all preceding views. □

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

Proof. Assume the contrary and let C_{fin} and C_{inf} be the sets of processes in $\mathcal{S}$ that execute the timer expiration handler finitely and infinitely often, respectively. Then $C_{\text{fin}} \neq \varnothing$, and by Claim 1 and Validity, $C_{\text{inf}} \neq \varnothing$. Let view v_2 be the first view such that $v_2 \geq \mathcal{V}$ and $E^{\mathcal{S}}_{\text{first}}(v_2) \geq \max\{\mathsf{GST}, t_{pj}\}$; such a view exists by Claim 1. The $\mathsf{timeout}$ value increases unboundedly at processes from C_{inf}, and does not change after some view v_3 at processes from C_{fin}. By Claim 1 and since leaders rotate round-robin, there exists a view $v_4 \geq \max\{v_2, v_3\}$ led by p_j such that any process $p_i \in C_{\text{inf}}$ that enters v_4 has $\mathsf{timeout}_i(v_4) > d + 3\Delta$. By Claim 1 and Validity, at least one process in $\mathcal{S}$ calls $\mathsf{advance}$ in v_4; let p_k be the first process to do so. Because $v_4 \geq v_3$, this process cannot be in C_{fin}, since none of these processes can increase their timers in v_4. Then $p_k \in C_{\text{inf}}$, contradicting Lemma 5.21. □

Since a process increases $\mathsf{timeout}$ every time $\mathsf{decision\_timer}$ expires, by Claim 2 all processes will eventually have

$$\mathsf{timeout} > d + 3(\delta + \rho)\mathsf{diameter}(\mathcal{S}).$$

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 $E^{\mathcal{S}}_{\text{first}}(v_1) \geq \max\{\mathsf{GST}, t_{pj}\}$ and for any process $p_i \in \mathcal{S}$ that enters v_1 we have

$$\mathsf{timeout}_i(v_1) > d + 3(\delta + \rho)\mathsf{diameter}(\mathcal{S}).$$

Then by Lemma 5.20, no process in $\mathcal{S}$ calls $\mathsf{advance}$ 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 $\mathsf{advance}$ in v_1, which is a contradiction. □

5.5 Possibility and Impossibility of Consensus via Ω

We now present some interesting consequences of our results. It is well known that the failure detector Ω [23, 24] (see §2.7.2) is sufficient for implementing wait-free consensus resilient to $\lfloor \frac{n-1}{2} \rfloor$ failures in non-partitionable systems subject to crash [24, 47], crash/recovery [8], or general omission [47, 60] failures. For instance, we presented one such implementation in §2.8. We now investigate whether this remains true under flaky channels and limited connectivity. As we show below, the answer depends on the amount of connectivity available in the underlying network, as established by our lower and upper bounds.

Lower bound. On the negative side, we first show that there exist weakly synchronous environments where Ω is implementable, but obstruction-free consensus is impossible even if at most one process may crash. One such environment is the system S of Aguilera et al. [6], where in every execution all channels may be flaky except those emanating from an a priori unknown correct process (the *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:

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

We now use Theorem 5.4 to show 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 $\mathcal{F}_{S,k} = \mathcal{F}_S \cap \{(P, C) \mid |P| \le k\}$. Then $\mathcal{F}_{S,k}$ is a k-fail-prone system.

Theorem 5.23. *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 for each $f \in \mathcal{F}_{S,k}$, $\tau(f) \neq \emptyset$. Then no algorithm can implement $(\mathcal{F}_{S,k}, \tau)$-obstruction-free consensus in $\mathcal{M}_{\text{PEF}}$.*

Proof. Suppose, for contradiction, that such an algorithm exists. A standard partitioning argument, as in Dwork et al. [32], implies that we must have $n > 2$ and $k \le \lfloor \frac{n-1}{2} \rfloor$. Since $\mathcal{M}_{\text{PRD}}$ is stronger than $\mathcal{M}_{\text{PEF}}$, Theorem 5.4 then requires that for all $f \in \mathcal{F}_{S,k}$, the graph $\mathcal{G} \setminus f$ contains a strongly connected component of size greater than $k \ge 1$. However, consider a failure pattern $f = (P, C) \in \mathcal{F}_{S,k}$ such that

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

In this case, $\mathcal{G} \setminus f$ consists only of trivial strongly connected components of size 1, contradicting the requirement above. Hence no such algorithm exists. $\square$

Upper bounds. On the positive side, we show that stronger connectivity assumptions do make consensus solvable in several previously proposed models of weak synchrony, including the systems S^+ and S^{++} of Aguilera et al. [6] and the models of [5, 33, 43, 51]. These works establish that Ω can be implemented in their respective settings, but – with the exception of [5] – do not provide a consensus algorithm.

To establish that consensus is possible in these models, we introduce an intermediate model $\mathcal{S}^\star$. In $\mathcal{S}^\star$, all channels may be asynchronous and flaky except those connecting an a priori unknown correct process (the *hub* [6]) to every other process in *both* directions; these hub channels are required to be eventually reliable, though still potentially asynchronous. In our framework, $\mathcal{S}^\star$ is captured as the set of executions of $\mathcal{M}_{\text{AEF}}$ that are 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 every $f \in \mathcal{F}_{\mathcal{S}^\star}$, the graph $\mathcal{G} \setminus f$ is strongly connected. Moreover, if at most $\lfloor \frac{n-1}{2} \rfloor$ processes fail in f, then $\mathcal{G} \setminus f$ is a connected core. Hence, by §4.4.2, we obtain:

Corollary 5.24. *A wait-free atomic register can be implemented in $\mathcal{S}^\star$ provided that at most $\lfloor \frac{n-1}{2} \rfloor$ processes may crash.*

Since wait-free consensus tolerating any number of $< n$ process crashes can be implemented using atomic registers together with Ω [48], Corollary 5.24 implies the following.

Corollary 5.25. *Wait-free consensus can be implemented in $\mathcal{S}^\star$ augmented with Ω, provided that at most $\lfloor \frac{n-1}{2} \rfloor$ processes may crash.*

We now use this result to prove the next corollary.

Corollary 5.26. *Wait-free consensus can be implemented over the systems S^+ and S^{++} of [6], as well as the models of [5, 33, 43, 51], provided that at most $\lfloor \frac{n-1}{2} \rfloor$ processes may crash.*

Proof. We first argue that the systems listed in the statement are at least as strong as $\mathcal{S}^\star$. In system S^+ [6], every execution has a *fair hub*, i.e., a correct process connected to all others by fair-lossy channels [3] in both directions. Since eventually reliable channels can be implemented on top of fair-lossy ones in an asynchronous system, S^+ is as strong as $\mathcal{S}^\star$. In S^{++} [6] and [5], every pair of processes is connected by fair-lossy channels, and thus S^{++} is also as strong as $\mathcal{S}^\star$. Finally, the models of [33, 43, 51] assume reliable channels between every pair of processes, making them strictly stronger than $\mathcal{S}^\star$. Since Ω can be implemented in all these models [5, 6, 33, 43, 51], Corollary 5.25 yields the result. □

Chapter 6

Conclusions

This thesis explored the foundations of fault-tolerant distributed computing under unreliable communication. Motivated by the practical challenges of message loss in modern networks, we considered a general model where both processes and channels may fail. In contrast to classical assumptions, we allowed *flaky* channels, which may drop messages arbitrarily without fairness or eventual delivery guarantees.

Our first contribution was a general characterization of solvability under arbitrary combinations of process and channel failures. To capture the minimal connectivity requirements for implementing classical shared-memory abstractions, we introduced the notion of a *generalized quorum system (GQS)*. We showed that a GQS characterize the necessary and sufficient connectivity assumptions for implementing atomic registers, snapshots, lattice agreement, and consensus. In particular, we proved that these problems can be solved even when none of the available read quorums are strongly connected – provided that some strongly connected write quorum is reachable from some read quorum. To match this lower bound, we presented algorithms for all four abstractions in a model with flaky channels. To solve atomic registers, we developed a new abstraction: *quorum access functions*, which leverage gossip-style communication and a novel *logical clock* to coordinate between indirectly connected quorums.

Having addressed this fundamental question, we then focused on a practically motivated class of fail-prone systems in which any minority of processes may fail. While this assumption restricts the class of admissible failure patterns, it yields stronger connectivity guarantees that can be exploited to design more efficient implementations – ones in which the time and memory required for an operation to complete can be bounded in terms of message delays and propagation. In this setting, we showed that consensus becomes the most challenging abstraction. To solve it, we adapted the *view synchronizer* abstraction, which enables a group of correct processes to spend sufficient time in a common view with a correct leader, without relying on traditional failure detectors such as Ω. Using this abstraction, we presented a consensus algorithm that tolerates flaky channels, works under partial synchrony, and uses bounded memory.

Taken together, our results show that the difficulty of solving classical problems in distributed computing depends strongly on the type of fault assumptions considered. In the general model, registers are the most difficult abstraction, and termination guarantees must be weakened. In more constrained settings, consensus becomes the hardest problem, but stronger termination and complexity guarantees become achievable.

Overall, this thesis contributes a general framework for reasoning about fault-tolerant distributed computing under unreliable communication, and provides new abstractions and techniques for designing provably correct algorithms in this setting.

6.1 Future Lines of Research

Several questions remain open. First, the upper bounds presented for shared-memory objects under a GQS incur high time and message complexity. It is worth investigating whether this cost is inherent, or whether more efficient algorithms can be developed under the same reliability assumptions.

Second, the upper bound for consensus under a GQS assumes that all correct channels are *eventually timely*. In the same way we established tight bounds for reliability, it would be interesting to determine whether one can also identify tight bounds on which subset of correct channels need to be eventually synchronous for consensus to be solvable.

Third, further research is required to understand the solvability and complexity of other classical distributed computing problems under arbitrary process and channel failures. Notably, problems such as k-set agreement, approximate agreement, renaming, k-exclusion, k-assignment, and crusader agreement remain unexplored in this generalized context. Determining their solvability conditions, lower bounds, and designing efficient algorithms for these problems will significantly enrich the theoretical foundations of fault-tolerant distributed computing.

Finally, future work may investigate whether the lower bounds identified here can be mitigated under additional system assumptions or alternative abstractions, such as probabilistic models of message loss or lightweight failure detectors. Empirical studies would also help assess the practical relevance of the proposed abstractions by identifying which failure patterns arise most often in real systems. Such patterns may differ from the $\frac{n}{2}$-fail-prone case, yet still provide stronger connectivity guarantees that can be exploited to design more efficient algorithms, thereby bridging the theoretical results with practical implementations.

References

[1] Atul Adya. “Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions”. Also as Technical Report MIT/LCS/TR-786. Ph.D. Cambridge, MA, USA: MIT, Mar. 1999.

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. “Atomic snapshots of shared memory”. In: *J. ACM* 40.4 (1993), pp. 873–890.

[3] 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”. In: *J. ACM* 41.6 (1994), pp. 1267–1297.

[4] Yehuda Afek and Eli Gafni. “Asynchrony from Synchrony”. In: *International Conference on Distributed Computing and Networking (ICDCN)*. 2013.

[5] 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.

[6] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. “On implementing Omega in systems with weak reliability and synchrony assumptions”. In: *Distributed Comput.* 21.4 (2008), pp. 285–314.

[7] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. “Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks”. In: *Theor. Comput. Sci.* 220.1 (1999), pp. 3–30.

[8] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. “Failure Detection and Consensus in the Crash-Recovery Model”. In: *Distributed Comput.* 13.2 (2000), pp. 99–125.

[9] 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.

[10] 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.

[11] Lăcrămioara Aştefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci-Piergiovanni, and Eugen Zălinescu. “Tenderbake - A Solution to Dynamic Repeated Consensus for Blockchains”. In: *Symposium on Foundations and Applications of Blockchain (FAB)*. 2021.

[12] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. “Sharing Memory Robustly in Message-Passing Systems”. In: *J. ACM* 42.1 (1995), pp. 124–142.

[13] 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.

[14] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. "Atomic snapshots using lattice agreement". In: *Distrib. Comput.* 8.3 (1995), pp. 121–132.

[15] Peter Bailis and Kyle Kingsbury. "The network is reliable". In: *Commun. ACM* 57.9 (2014), pp. 48–55.

[16] 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.

[17] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. "Liveness and latency of Byzantine state-machine replication". In: *Symposium on Distributed Computing (DISC)*. 2022.

[18] Manuel Bravo, Gregory Chockler, and Alexey Gotsman. "Making Byzantine consensus live". In: *Distributed Comput.* 35.6 (2022), pp. 503–532.

[19] Eric A. Brewer. "Towards robust distributed systems (abstract)". In: *Symposium on Principles of Distributed Computing (PODC)*. 2000.

[20] Marc Brooker, Tao Chen, and Fan Ping. "Millions of Tiny Databases". In: *Symposium on Networked Systems Design and Implementation (NSDI)*. 2020.

[21] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. 2nd. Springer Publishing Company, Incorporated, 2011.

[22] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Errata for Introduction to Reliable and Secure Distributed Programming*. https://www.distributedprogramming.net/docs/Errata.pdf. Accessed: July 23, 2025.

[23] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. "The weakest failure detector for solving consensus". In: *J. ACM* 43.4 (1996), pp. 685–722.

[24] Tushar Deepak Chandra and Sam Toueg. "Unreliable Failure Detectors for Reliable Distributed Systems". In: *J. ACM* 43.2 (1996), pp. 225–267.

[25] Gregory Chockler, Idit Keidar, and Roman Vitenberg. "Group Communication Specifications: A Comprehensive Study". In: *ACM Comput. Surv.* 33.4 (2001), pp. 427–469.

[26] Brian Coan. "A Compiler That Increases the Fault Tolerance of Asynchronous Protocols". In: *IEEE Trans. Comput.* 37.12 (1988), pp. 1541–1553.

[27] Étienne Coulouma and Emmanuel Godard. "A Characterization of Dynamic Networks Where Consensus Is Solvable". In: *Structural Information and Communication Complexity (SIROCCO)*. 2013.

[28] Étienne Coulouma, Emmanuel Godard, and Joseph Peters. "A characterization of oblivious message adversaries for which Consensus is solvable". In: *Theoretical Computer Science* 584 (2015), pp. 80–90.

[29] Danny Dolev. "The Byzantine Generals Strike Again". In: *J. Algorithms* 3.1 (1982), pp. 14–30.

[30] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. *Failure detectors in omission failure environments*. Tech. rep. TR96-1608. Department of Computer Science, Cornell University, 1996.

[31] 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.

[32] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. “Consensus in the presence of partial synchrony”. In: *J. ACM* 35.2 (1988), pp. 288–323.

[33] Antonio Fernández Anta and Michel Raynal. “From an Intermittent Rotating Star to a Leader”. In: *Conference on Principles of Distributed Systems (OPODIS)*. 2007.

[34] Faith Fich, Maurice Herlihy, and Nir Shavit. “On the Space Complexity of Randomized Synchronization”. In: *J. ACM* 45.5 (1998), pp. 843–862.

[35] Colin Fidge. “Timestamps in message-passing systems that preserve the partial ordering”. In: *Australian Computer Science Conference (ASCS)*. 1988.

[36] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382.

[37] Roy Friedman, Idit Keidar, Dahlia Malkhi, Ken Birman, and Danny Dolev. *Deciding in partitionable networks*. Tech. rep. TR95-1554. Department of Computer Science, Cornell University, 1995.

[38] Seth Gilbert and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: *SIGACT News* 33.2 (2002), pp. 51–59.

[39] Vassos Hadzilacos. *Byzantine agreement under restricted type of failures (not telling the truth is different from telling lies)*. Tech. rep. TR-18-63. Department of Computer Science, Harvard University, 1983.

[40] Maurice Herlihy. “Wait-Free Synchronization”. In: *ACM Transactions on Programming Languages and Systems* 13.1 (1991), pp. 124–149.

[41] 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.

[42] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. “Flexible Paxos: Quorum intersection revisited”. In: *Conference on Principles of Distributed Systems (OPODIS)*. 2016.

[43] Martin Hutle, Dahlia Malkhi, Ulrich Schmid, and Lidong Zhou. “Chasing the Weakest System Model for Implementing Ω and Consensus”. In: *IEEE Trans. Dependable Secur. Comput.* 6.4 (2009), pp. 269–281.

[44] 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.

[45] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (1978), pp. 558–565.

[46] Leslie Lamport. “On Interprocess Communication - Part I: Basic Formalism, Part II: Algorithms”. In: *Distributed Comput.* 1.2 (1986), pp. 77–101.

[47] Leslie Lamport. “The Part-Time Parliament”. In: *ACM Trans. Comput. Syst.* 16.2 (1998), pp. 133–169.

[48] 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.

[49] Nancy Lynch. “Distributed Algorithms”. In: Morgan Kaufmann, 1996. Chap. 17.

[50] Nancy A. Lynch and Alexander A. Shvartsman. “RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks”. In: *Symposium on Distributed Computing (DISC)*. 2002.

[51] 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.

[52] Dahlia Malkhi and Michael K. Reiter. “Byzantine Quorum Systems”. In: *Distributed Comput.* 11.4 (1998), pp. 203–213.

[53] Moni Naor and Avishai Wool. “The load, capacity and availability of quorum systems”. In: *Symposium on Foundations of Computer Science (FOCS)*. 1994.

[54] Oded Naor, Mathieu Baudet, Dahlia Malkhi, and Alexander Spiegelman. “Cogsworth: Byzantine View Synchronization”. In: *Cryptoeconomics Systems Conference (CES)*. 2020.

[55] Oded Naor and Idit Keidar. “Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR”. In: *Symposium on Distributed Computing (DISC)*. 2020.

[56] Gil Neiger and Sam Toueg. “Automatically Increasing the Fault-Tolerance of Distributed Algorithms”. In: *J. Algorithms* 11.3 (1990), pp. 374–419.

[57] Harald Ng, Seif Haridi, and Paris Carbone. “Omni-Paxos: Breaking the Barriers of Partial Connectivity”. In: *European Conference on Computer Systems (EuroSys)*. 2023.

[58] Thomas Nowak, Ulrich Schmid, and Kyrill Winkler. “Topological Characterization of Consensus under General Message Adversaries”. In: *Symposium on Principles of Distributed Computing (PODC)*. 2019.

[59] Kenneth J. Perry and Sam Toueg. “Distributed agreement in the presence of processor and communication faults”. In: *IEEE Trans. Software Eng.* 12.3 (1986), pp. 477–482.

[60] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. “Revisiting the PAXOS algorithm”. In: *Theor. Comput. Sci.* 243.1-2 (2000), pp. 35–91.

[61] Dimitris Sakavalas and Lewis Tseng. *Network Topology and Fault-Tolerant Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2019.

[62] Nicola Santoro and Peter Widmayer. “Time is not a healer”. In: *Symposium on Theoretical Aspects of Computer Science (STACS)*. 1989.

[63] Nicola Santoro and Peter Widmayer. “Distributed function evaluation in the presence of transmission faults”. In: *Symposium on Algorithms (SIGAL)*. 1990.

[64] Ulrich Schmid, Bettina Weiss, and Idit Keidar. “Impossibility Results and Lower Bounds for Consensus under Link Failures”. In: *SIAM Journal on Computing* 38.5 (2009).