Politecnico di Torino

Master's Degree in Mathematical Engineering

Academic Year 2024/2025

Consensus Algorithms for Distributed Systems: Managing the Consistency of Critical Data Files

Supervisors:

Antonio J. Di Scala

Fadi Barbàra

Candidate:

Ilaria Palumbo

Table of Contents

Introduction

Most of the services that we use are based on a fundamental assumption that is often taken for granted, namely trust. Every day, we rely on service providers or, more broadly, on institutions that manage our assets or data, trusting that they will act correctly. These services are based on centralized systems in which a single entity controls, coordinates, and validates every operation, acting as a central authority. An emblematic example is the current financial system, which relies on a set of banking institutions to which we entrust the custody and management of our assets. Although banks provide us with valuable services, using them means placing our trust in a system whose security and correctness depend entirely on a central authority. In this case, that authority is a central bank. While relying on such an institution may seem natural and reasonable, it actually exposes us to significant risks. Financial crises and inflation show the consequences that a failure of such an authority could have, and highlight the value of the trust we place in this system. The same reasoning applies to any centralized system that provides a service: an error, a failure or a malicious behaviour on the part of the central authority can compromise the entire system.

The aim of distributed systems is to create a network of agents that collaborate to provide a service without relying on a single entity. These agents can be imagined as individuals who do not know each other and therefore have no reason to trust one another, yet still wish to cooperate towards a shared objective. The core challenge that distributed systems aim to address is ensuring the system behaves correctly in the absence of pre-existing trust. The basic idea is to establish a set of common rules that allow agents to cooperate directly without mutual trust or the need for a central authority. When applied to a computer network, this approach results in systems in which multiple machines collaborate to achieve a common goal. This requires the machines to coordinate their actions and maintain a coherent view of the system's state. This leads to several advantages: the system becomes more resilient, since the failure of a single machine does not affect the operation of the whole system, and it becomes more transparent, since all components follow the same set of rules that define the system's behaviour, namely a protocol. Consensus protocols are used when components need to make collective decisions, such as agreeing on the state of the system or determining which operations to perform; in other words, whenever they must reach a consensus.

This thesis focuses on the application of distributed systems to the management of critical files containing sensitive data. It is crucial that this data is not lost and is updated correctly when necessary. The aim is to exploit the fault tolerance of distributed systems by replicating files across multiple servers rather than relying on a single central server, whose failure could result in data loss. This introduces the challenge of ensuring data consistency, maintaining identical copies and applying updates coherently across the system. To achieve this, the system relies on a consensus algorithm.

Contributions

In this thesis, I created a new consensus algorithm to satisfy the peculiar characteristics of the system under study. This system features an architecture that combines distributed and centralized elements. The main objective was to understand how a consensus algorithm could operate in this context.
To understand and appreciate the differences between the consensus algorithms in the literature and mine, I provide an investigation of traditional consensus algorithms. This investigation also clarifies the assumptions and strategies that enable a consensus algorithm to be implemented in practice. These insights were essential for the design phase of the proposed algorithm.
In this work, I also provide a reference implementation of my consensus algorithm. The code is available on GitHub, in the public repository `distributed-consensus-`

`star-topology` [1].
Furthermore, I provide an extension to consensus algorithms that incorporates reputation systems. This extension shows how a reputation system can support the decision process of my consensus algorithm.

Overview of the Thesis

The thesis is organized into six chapters.
Chapter 1 introduces distributed systems. It presents the fundamental concepts and reference models that serve as the foundation for the subsequent chapters. Chapter 2 investigates consensus algorithms in distributed systems. It provides the conceptual tools required to understand the proposed Consensus Algorithm and the assumptions that support its design. Chapter 3 introduces reputation systems. It provides the general background needed to understand their role and their subsequent integration in the consensus protocol. Chapter 4 provides the proposed Consensus Algorithm. It explains how the algorithm is designed to meet the architectural constraints of the case study and how its logic is validated through simulation. Chapter 5 provides an extension of the Consensus Algorithm based on reputation. It illustrates how a reputation system can be integrated into the consensus protocol to support its decision process. Chapter 6 discusses the main limitations of this work and suggests several directions for future work.
Finally, Appendix A contains the source code of the consensus module.

[1] `https://github.com/ilariapalumbo/distributed-consensus-star-topology.git`

Chapter 1

Foundations of distributed systems

Distributed systems emerged thanks to technological advances in hardware and network components, which made it possible to create computer networks. They offer an alternative to centralized systems and address their inherent limitations in terms of reliability and scalability.

This chapter introduces the basic notions of distributed systems. It presents the main properties, with particular attention to reliability and consistency. Next, the State Machine Replication model is introduced as it forms the basis for understanding how multiple machines can maintain consistent states. The chapter also discusses the CAP theorem, a foundational result in distributed systems that is useful for reasoning about the level of consistency achievable in real systems.

1.1 Definition and basic concepts

According to Tanenbaum and van Steen, a *distributed system* is a collection of independent computing elements that appears to its users as a single coherent system [18]. In this work, these computing elements are referred to as *nodes*. These nodes can represent either hardware components or software processes, and the definition makes no assumptions about how they are interconnected.

The defining feature of these systems is that, although each node operates independently, they must cooperate to achieve a common goal, enabling the system to behave as a single entity. This feature distinguishes distributed systems from centralized ones, in which a single node – the *central authority* – coordinates all operations. In this sense, it maintains the global control of the system. In a distributed system, however, such an authority does not exist and coordination is achieved through direct interactions between nodes instead.
Distributed systems rely on *protocols*, well-defined sets of rules that all nodes must follow to achieve cooperation. Following the same protocol enables autonomous nodes to communicate with each other and make collective decisions.

The design of these systems is inherently complex. They must preserve properties such as resource sharing, distribution transparency, openness, scalability [18] and reliability. In this work, we will focus on reliability.

Finally, since a distributed system consists of multiple nodes, it must also manage *group membership.* A system may adopt an open group model, in which any node can join the network, or a closed group model, in which only known and authorized members are permitted to participate.

Network structures

A distributed system is characterized by a network of nodes that does not have a single central node. In this setting, nodes can interact directly with one another. To illustrate this, we compare the structure with centralized and decentralized network models.

A *centralized network* is characterized by the presence of a single central node that serves as a connection point for all other nodes (Figure **a** in 1.1). This node represents a global point of control, since the coordination of operations depends entirely on it. Consequently, it also becomes a *single point of failure* for the entire system.

A *decentralized network* includes several nodes that act as local centres of control. Each of these nodes manages a cluster of subordinate nodes and maintains links with the other centres (Figure **b** in 1.1).

Finally, a *distributed network* has no central point of control: each node can communicate directly with any other, without a predefined hierarchy (Figure **c** in 1.1). In this configuration, the control of the system is shared equally among all nodes.

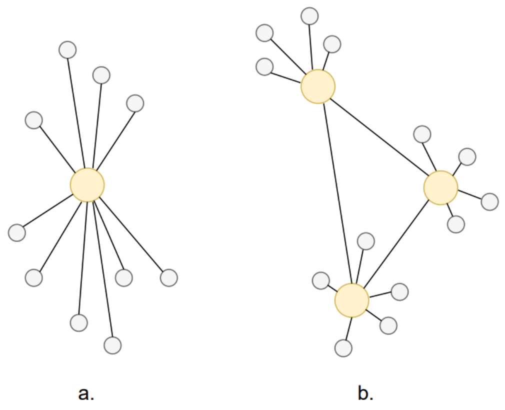

Figure 1.1: Representation of: **a.** centralized network; **b.** decentralized network; **c.** distributed network.

Architectural models

We will now present the main architectural models. The term *architecture* refers to the logical organization of nodes and their interaction patterns.

In a *client–server* architecture there are two distinct roles: the server, which provides a service, and the client, which issues service requests. This model is naturally asymmetric, since clients and servers perform different functions. In its traditional form, it relies on a centralized network, where a central server manages requests from multiple clients. In this sense, the server acts as a coordinator of the service.

In distributed implementations, however, multiple servers may operate in parallel while maintaining the same logical distinction between clients and servers. The model can also evolve into more distributed forms, such as multi-tiered architectures, where the same logic is divided into several interacting layers [18].

In a *peer-to-peer* architecture, the distinction between client and server disappears. Each node, or peer, can act as both a provider and a requester of a service, interacting with other peers symmetrically. This model usually relies on a distributed network.

A *hybrid* architecture combines features of the two previous models. Some nodes may act as coordinators while others operate as ordinary peers. This architecture is typically associated with decentralized networks.
This hybrid form is adopted by many real systems because it provides an effective compromise between centralized and distributed solutions.

Synchronization

In distributed systems, time plays a fundamental role in determining how independent nodes can coordinate their actions.
Each node operates according to its own local notion of time, with no shared temporal reference. Consequently, *synchronizing* processes is one of the typical issues that a distributed system must address.

In *Time, Clocks, and the Ordering of Events in a Distributed System* [13], Leslie Lamport examined the notion of time in distributed environments by introducing an ordering relation among events. In particular, the *happened-before* relation captures causality: if an event a can influence an event b, then we write $a \to b$. However, this relation defines only a partial order, since there may be concurrent events that are not causally related.

To extend this relation to a *total order*, Lamport introduced a system of *logical clocks* that assigns a timestamp C to each event, consistent with causality, such that $a \to b \Rightarrow C(a) < C(b)$.

The resulting total ordering forms the basis of the *State Machine Replication* model, in which all components execute operations in the same order to maintain a consistent system state (see section 1.2.1 for details). This notion of ordering is also essential for defining the different *consistency models* that will be discussed later.

Communication models

In a distributed system, nodes collaborate by exchanging messages.
There are three classical communication models that are commonly distinguished:

- *Synchronous*: message delivery times are bounded and known.
- *Asynchronous*: message delivery times are unbounded, meaning a message could be delayed indefinitely.
- *Partially synchronous:* message delivery times are bounded but the upper bound is unknown.

The concept of partial synchrony, introduced by Dwork, Lynch, and Stockmeyer [2], lies between the synchronous and asynchronous extremes. In real systems, messages are usually delivered after an unknown but finite amount of time. The partial synchrony model captures this behaviour by assuming the existence of a *Global Stabilization Time (GST)*, after which the delivery time of messages becomes known. Therefore, partial synchrony implies that the system is asynchronous and will eventually stabilize, becoming temporarily synchronous.
Partial synchrony provides a foundation for designing practical consensus protocols, as discussed later in Chapter 2.

Properties and objectives

One of the main goals of a distributed system is to ensure *reliability*, meaning the system's ability to continue operating without interruption over time. To achieve this, the system must be *fault-tolerant*, meaning it must continue to operate correctly even in the presence of faults[1].

Failures in distributed systems are usually *partial*: a fault in one component may affect only a subset of nodes, with the rest of the system continuing to function normally. This behaviour contrasts with that of centralized systems, where a failure often results in a complete loss of functionality.

Fault tolerance is typically achieved through *redundancy*, whereby the system can compensate for component failures. When a component fails, it can be isolated or replaced by a redundant component to ensure the system continues to operate. Redundancy can be introduced in several ways:

- Time redundancy: repeating operations so that, if a component fails temporarily, the operation can succeed once it recovers.
- Physical redundancy (or replication): duplicating hardware components or software processes across different nodes so that, if one fails, another can replace it.

Data replication is a specific form of replication, whereby the content managed by one node is replicated across other nodes in the network; the resulting copies are hereafter referred to as replicas.
This technique is widely used in distributed systems to improve not only reliability, but also performance and scalability by distributing the workload across multiple nodes. If data were stored on a single server, concurrent requests could overwhelm it, resulting in a performance bottleneck. However, by distributing replicas across different nodes, the system can handle a larger number of requests in parallel, thus increasing its overall throughput. Furthermore, by placing replicas closer to the processes that access them, the system can reduce response times, which is an important advantage in geographically distributed environments where network latency is significant.

[1]In this thesis, we use the term *fault* to refer to the cause (e.g. a crash) of a *failure*, i.e. a malfunction of the system.

However, these benefits come at the cost of greater complexity in maintaining data consistency among replicas. Whenever one replica is modified, all the others must eventually reflect the same change in order to preserve a coherent global state. Achieving this consistency requires coordination among distributed nodes, which often involves additional message exchanges or specific protocols.

Finally, redundancy for fault tolerance is also related to *availability*, meaning the system's ability to remain operational at any time, even in the event of failures. By maintaining multiple replicas, a system can continue to serve requests even if some components become unavailable by redirecting them to the functioning replicas.
Availability captures service continuity at a specific point in time, whereas reliability refers to the ability to operate correctly and without interruptions over time.

Fault models

In order to design a reliable distributed system, it is important to understand the different types of faults that may occur and how they affect system behaviour:

- Crash fault: a component stops working.
- Omission fault: a component does not respond.
- Timing fault: a component responds too late.
- Response fault: a component produces an incorrect output.
- Byzantine fault: a component behaves unpredictably or even maliciously.

Byzantine faults are the most difficult to tolerate. In such cases, faulty components may deviate arbitrarily from the protocol, or even collude with others to produce strategically false results.

In addition to process failures, distributed systems must also cope with the unreliability of the network. Messages may be delayed or lost, and the network may partition into groups of nodes that can no longer communicate with each other. In this case, the fault lies not in the nodes, but in the communication links.

Network partitions have significant implications for the behaviour of the system, which are examined by the CAP theorem, a fundamental result in distributed systems theory (see section 1.3).

1.2 Fundamental models and properties

1.2.1 The State Machine Replication model

The *State Machine Replication* (SMR) model, introduced by Schneider [16], provides a general framework for building fault-tolerant distributed services.
It assumes a distributed client–server architecture, in which several replicas of the same server run on different nodes. However, to make the system behave as if there were a single logical server, client requests must be processed in a coordinated manner across the replicas.

Each replica implements a deterministic *state machine*, which is defined by a set of state variables representing its current state, and a collection of commands that modify this state in response to client requests. Since each state machine is deterministic, if all replicas start from the same initial state and execute the same sequence of commands, they will produce identical outputs and provide consistent responses to clients.
The SMR approach therefore requires commands to be executed in a common total order on all replicas. Reaching and maintaining such an order requires coordination among nodes, typically achieved through consensus protocols (see Chapter 2 for details).

Fault tolerance in SMR arises from redundancy and determinism: since all replicas perform the same operations, any replica producing a different result can be identified as faulty and disregarded. This mechanism ensures that all non-faulty replicas maintain identical states, thereby guaranteeing *consistency* among them.

In general, consistency is a property that distributed systems aim to preserve. Its precise meaning depends on the context, but it generally refers to the ability

of all nodes to maintain a coherent view of the system state. By preserving this coherence, the system behaves as if it were a single entity.
Depending on the design goals of the system, different levels of consistency may be required, resulting in distinct consistency models, which are introduced in the next section.

1.2.2 Consistency models

As mentioned, the problem of consistency arises when data are replicated across multiple nodes. This issue is usually considered in relation to *read* and *write* operations on shared data: write operations modify the data stored in a given node, and this change must be propagated to all replicas; meanwhile, read operations retrieve the local replica.
Consistency models define the guarantees that a system provides regarding how write operations are observed across nodes, and what read operations return.

Atomic consistency

Atomic consistency, also known as *linearizability*, is the strongest and most intuitive form of consistency. In this model, all operations appear to be executed instantaneously in a common total order. Consequently, every read operation returns the value written by the most recent write operation.
Such a guarantee requires the establishment of a common total order of operations that respects real-time ordering so that the effect of each operation is immediately visible to all nodes. In practice, however, achieving this level of consistency is challenging and computationally expensive, which is why real systems often rely on weaker models.

Sequential consistency

Sequential consistency, as defined by Lamport [11], relaxes the real-time constraint. In this model, all nodes observe operations in the same total order, even if this order does not correspond to real-time execution.
A common total order is defined over all operations, forming a logical sequence that preserves the internal order of each process, though not necessarily the temporal

order in which operations actually occur. Consequently, a read operation may return an outdated value because some write operations are not yet visible in the logical sequence at that point in time.

Eventual consistency

Eventual consistency is a weaker form of consistency that is commonly used in systems that require few write operations or where these operations are performed by a single node. Unlike previous models, it does not require nodes to apply operations in the same order. Consequently, replicas may diverge temporarily, but, provided there are no new writes, they will eventually converge to the same state. Eventual consistency is widely adopted in large-scale distributed systems where maintaining global coordination would be too costly.

1.2.3 System correctness: Safety and Liveness properties

In distributed systems, *correctness* is defined in terms of the properties that describe how the system should behave. There are two main classes of properties used to characterize correct behaviour: *Safety* and *Liveness*.

A safety property ensures that nothing bad ever happens, meaning that the system never produces incorrect results. A liveness property, instead, ensures that something good eventually happens, meaning that the system continues to make progress and eventually gives a response. In short, safety concerns the correctness of the system state, while liveness relates to its ability to make progress.

In unreliable environments, where nodes or communication channels may fail, ensuring both safety and liveness is particularly challenging. The study of the trade-off between these two properties has led to several *impossibility results* in the field of distributed computing. The following sections discuss two such results: the CAP theorem (section 1.3) and the FLP result (section 2.2).

1.3 The CAP theorem

In 2000, Eric Brewer introduced the idea that it is impossible for distributed systems to guarantee *Consistency*, *Availability* and *Partition tolerance* (CAP). This observation was later formalized and proven by Gilbert and Lynch [7], and became known as the CAP theorem.
Although Brewer initially discussed the problem in the context of distributed web services more broadly, the formalization relies on a distributed shared memory model supporting read and write operations.

The three properties

According to Brewer's formulation, an ideal distributed service should satisfy three properties:

- Consistency: corresponds to *linearizability*, introduced in Section 1.2.2. It requires all nodes to observe operations in the same total order, so that every read returns the result of the most recent write.
- Availability: every request addressed to a non-faulty node must eventually receive a valid response. This implies that the system remains responsive even when some components fail or become temporarily unreachable.
- Partition tolerance: the system continues to operate correctly even if the network is split into disjoint subsets that cannot communicate with each other.

In practice, any distributed system should provide partition tolerance, since real networks are unreliable and messages may be delayed or never delivered. This behaviour reflects asynchronous communication, meaning that network partitions are a manifestation of asynchrony and an inherent feature of real systems.

The impossibility result

Using the formalization of Gilbert and Lynch, we can now formulate the impossibility result for a distributed service that provides read and write operations on a shared memory. However, the same reasoning applies to any distributed service that can be abstracted in terms of these basic operations.

Theorem (CAP): *In an asynchronous network, it is impossible to implement a service that can guarantee both availability and consistency in all fair executions, including those affected by the loss of messages.*

Intuitively, the impossibility arises when a network partition isolates part of the system.
For example, if a client sends a read request to a node that is disconnected from the one that processed the last write, the former cannot determine whether the data has been modified. To preserve consistency, the node should wait until communication is restored. However, if the partition persists, the node may wait indefinitely, which would violate the availability requirement. Alternatively, if it responds immediately, it may return an outdated value, thus violating consistency. Therefore, during a partition, a distributed system must sacrifice either availability or consistency.

Interpretation of the result

Since network partitions are unavoidable in real distributed systems, the practical trade-off lies between consistency and availability.
However, in the CAP framework, both properties are defined in their strongest forms (linearizability and full availability) which few real systems can actually achieve. Furthermore, the assumptions underlying the theorem limit its applicability. The theorem is based on an asynchronous network model in which communication failures are represented as network partitions. It does not account for faulty nodes, although a node crash can be interpreted as a trivial partition that isolates a single node.

For these reasons, Gilbert and Lynch interpret the CAP theorem as a specific instance of a broader principle in distributed computing: the impossibility of guaranteeing both safety and liveness in unreliable systems.
In this analogy, consistency corresponds to a safety property (every response must be correct) and availability corresponds to liveness (every request eventually receives a response). Unreliability arises from network partitions.

Despite its limitations, the CAP theorem remains a valuable conceptual tool. It provides an intuitive framework for reasoning about design trade-offs in distributed systems, even when their actual behaviour differs from the strict assumptions of the theorem. An application of this reasoning will be discussed in Chapter 4, Section 4.2.2.

Chapter 2

Consensus algorithms

In the previous chapter, we introduced the idea that nodes in a distributed system must cooperate according to a common set of rules: a protocol. The communication protocol defines how nodes can exchange messages, but message exchange alone does not guarantee that they will make consistent decisions towards a common goal. To coordinate their actions and maintain a consistent system state, distributed systems rely on consensus protocols. These protocols enable nodes to agree on a specific outcome, allowing the system to function as a coherent whole.

Reaching agreement in an unreliable distributed system, where crashes, delays, or network partitions may occur, is not straightforward and it is known as the consensus problem. This chapter first introduces this problem. It then examines a fundamental theoretical result that establishes the conditions under which consensus can be achieved. This result serves as a reference for the design of practical consensus protocols and provides the theoretical basis for the work developed in Chapter 4. Finally, the chapter outlines the main characteristics of practical consensus protocols, focusing in particular on the Raft algorithm, which has been adopted as the reference model for the consensus protocol designed and implemented in Chapter 4.

2.1 The consensus problem

Achieving consensus in a distributed system means reaching a common decision. Nodes must agree on a specific outcome, whose nature depends on the system's purpose: it may represent an action to perform, a data item to validate, or simply a numerical value.

To illustrate this idea, consider the *distributed transaction commit problem.*
In a distributed database system, a transaction is a set of operations executed across multiple nodes. The transaction completes when one node, acting as the coordinator, requests all other nodes to either commit or abort the transaction. The transaction is successfully committed only if all nodes agree to commit; if even a single node votes to abort, the entire transaction must be aborted.

This simple mechanism captures the essence of *distributed consensus*: independent components must reach a common decision to maintain a consistent global state.

The decision process is coordinated through a protocol. In this example, the classical approach is the Two-Phase Commit (2PC), which relies on a simple exchange of acknowledgements. The coordinator proposes to commit a transaction and requests confirmation from all participants. If it receives a positive response from every node, the transaction is committed and applied everywhere. This protocol ensures the *atomicity* of distributed operations: all nodes either commit the transaction or none of them do.

This example shows that reaching agreement is straightforward when no failures or communication issues occur. However, once such faults are considered, the problem becomes significantly more complex. For instance, in the Two-Phase Commit protocol, if the coordinator crashes during the process, the participants cannot determine whether the transaction should be committed or aborted, and thus the consensus cannot be reached.

A well-known example that illustrates this difficulty is the *Byzantine Generals*

Problem. Introduced by Lamport et al.[14], this metaphor describes the challenge of achieving agreement in a distributed system subject to Byzantine faults, condition under which components may behave arbitrarily or even maliciously.

The problem. A group of generals of the Byzantine army is camped around an enemy city. Each general commands a division, and all divisions must agree on a common plan: either to attack or to retreat. The success of the operation depends on coordination: if only part of the army attacks, the assault will inevitably fail.
The generals can communicate only by sending messages. However, some generals may be traitors and communication channels are unreliable: messages can be delayed, lost, intercepted, or forged by dishonest parties.
Consider, for example, a scenario with five generals. Two of them plan to attack, and two plan to retreat. If the fifth general acts maliciously and sends conflicting orders, telling some to attack and others to retreat, the army becomes divided. Although each general may believe an agreement has been reached, in reality, no consistent decision has been made: one part attacks while the other withdraws, and the plan fails.

From the problem description emerges the intrinsic complexity of achieving agreement in an *unreliable* environment.
Lamport et al. [14] formalized the problem by introducing a structured setting with two types of participants: a commander, who issues the order, and a group of lieutenants, who must agree on how to act based on the messages they receive.
The goal is to ensure that all non-faulty lieutenants reach the same decision, and that this decision reflects the commander's order whenever the commander is not faulty.

This formulation captures the essence of the problem: correct participants must agree on a single decision despite the communication system is unreliable and there are potentially malicious nodes.
Lamport et al. proposed several algorithms to address this issue, the first of which, Oral Messages (OM), relies on multiple rounds of message exchange among the

generals to ensure that correct nodes can eventually agree. Although conceptually simple, this approach results in high communication overhead as the number of faults increases.

This example shows why consensus protocols[1] are essential: they provide a mechanism for nodes to reach agreement despite the presence of faults or communication issues.
The complexity of the problem and the protocol depend on the failure model: crash-tolerant protocols are simpler to implement, while Byzantine fault-tolerant ones are considerably more difficult.

Formalization of the problem

Now we introduce a simple formalization of the consensus problem, consistent with the classical literature and useful for the discussion in the following section. In this formalization, the nodes of a distributed system are modelled as *processes*, each executed on a separate node.

The consensus problem involves a set of n processes $p_1, \dots, p_n$ that communicate by exchanging messages. Each process p_i initially holds a value $v_i \in V$, where V is the set of possible values. The goal is for all correct processes to reach a common decision value v, satisfying the following properties:

- Agreement: all correct processes decide on the same value.
- Validity: every decision value must have been the initial value of some process.
- Termination: every correct process eventually decides.

A consensus protocol must satisfy these properties even in the presence of faults.

[1]In this context, the term consensus protocol refers to a *distributed consensus algorithm*. Unlike centralized settings, where a single coordinator can enforce the decision, distributed nodes communicate only through message exchange. This absence of global coordination, together with possible failures, makes distributed consensus a fundamental and challenging problem in distributed computing.

Beyond fault tolerance, one of the main difficulties in achieving consensus lies in satisfying all three properties simultaneously. In particular, the termination property raises the question of when consensus can actually be reached – that is, whether the consensus problem admits a solution.
For instance, the solution to the Byzantine Generals Problem relies on an implicit but crucial assumption: the system is synchronous, meaning that message transmission and processing times are bounded and known. This assumption is essential to ensure that the consensus algorithm eventually terminates and a decision is reached. When such an assumption does not hold, as we will see in the next section, reaching agreement can no longer be guaranteed.

2.2 The Fischer-Lynch-Paterson result

The result provided by Fischer, Lynch and Paterson (FLP) represents a fundamental limit in the theory of distributed systems [4]. It demonstrates that, even under the weakest fault model where a single process may crash, it is impossible to guarantee that distributed consensus is always reachable in an asynchronous system.

The authors proved their result in a simple setting, so that the impossibility would also hold in more complex or adverse environments. To this end, they considered a system model based on the following assumptions.
The first and most important assumption is *full asynchrony*: there are no bounds on message transmission times or process execution speeds. In particular, processes have no synchronized clocks and therefore cannot progress in lockstep or at a known fixed rate. A fundamental implication of full asynchrony is that processes cannot distinguish between a failed node and a slow one – that is, a process that is still operational but taking an arbitrarily long time to do its processing and to send messages.
Second, the system is assumed to be free of Byzantine faults: processes can only experience crash faults, meaning that they may stop executing at any time.
Finally, communication channels are assumed to be reliable: every message sent is eventually delivered correctly.

The authors also focused on the simplest instance of the consensus problem, known as binary consensus, where decision values belong to the set $V = 0, 1$.

Before presenting the impossibility result, we introduce some basic notions that will be used in its formalization.

- configuration c: the state of the overall system, including, for each process p_i, its current value v_i and the set of messages it has received.
- event e: the action of a single process p_i, consisting in receiving a message, updating its local value v_i, and possibly sending new messages.
- run r: a finite or infinite sequence of events that drives the system from one configuration to another.

A consensus protocol is said to be *partially correct* if it satisfies two conditions:

1. Uniqueness of the decision value, corresponding to the agreement property introduced in Section 3.1.
2. Each decision value can occur in some reachable configuration. This condition excludes trivial protocols that always decide the same fixed value, and corresponds to the validity property discussed in Section 3.1.

In other words, partial correctness guarantees that decisions are correct, but not that they will actually be reached.

A protocol is *totally correct* if:

1. it is partially correct,
2. in every admissible run (i.e., with at most one faulty process) at least one process eventually reaches a decision. This corresponds to the termination property introduced in Section 3.1.

Hence, total correctness ensures that a decision is always reached, while partial correctness only guarantees the correctness of possible decisions.

The impossibility result

Theorem (FLP): *In a fully asynchronous distributed system, no consensus protocol can be totally correct, even in the presence of a single crash fault.*

This means that in an asynchronous system it is impossible for a consensus protocol to guarantee both agreement and termination.

The proof is based on the notion of *bivalence.*
A protocol can start from a bivalent configuration (a system state in which both decision values remain possible during a run), as opposed to a univalent configuration, where every admissible run leads to the same decision value. From such a bivalent configuration, it is possible to construct an infinite run: a sequence of events that keeps the system bivalent indefinitely.
This behaviour arises directly from the asynchronous nature of the system. Since message delays are unbounded, a process may take an arbitrarily long time to respond, and the others can never determine whether it is simply slow or has crashed. Even the possibility of a single crash is enough to introduce this uncertainty.
To preserve the agreement property, correct processes cannot decide until they are certain of the global state. In an asynchronous system, this may never happen. As a consequence, the system can remain bivalent forever, never reaching a decision. Thus, even with only one faulty process, there always exists at least one admissible run in which the protocol never terminates.

The FLP result is one of the foundational impossibility theorems in distributed systems. It reveals a fundamental trade-off between safety and liveness, the same classes of properties introduced in the previous chapter.
In a consensus protocol, agreement and validity correspond to safety properties, meaning that no two processes decide differently, and every decision value is valid. Termination, instead, is a liveness property, as it guarantees that every process eventually decides. The impossibility proved by Fischer, Lynch and Paterson therefore shows that no consensus protocol can ensure safety and liveness simultaneously in a fully asynchronous system.

This result is closely related to the CAP theorem (see Chapter 1), since both express a similar limitation: the impossibility of achieving consistency (safety) and availability (liveness) at the same time in unreliable asynchronous environments.

Interpretation of the result

The FLP result does not imply that distributed consensus is always unreachable, but rather that it cannot be guaranteed under the assumptions of a fully asynchronous system. In practice, the termination of a consensus protocol can be achieved under specific conditions.

Fischer, Lynch, and Paterson themselves showed that there exists a partially correct consensus protocol that allows all correct processes to eventually decide, provided that no failures occur during execution and that a strict majority of processes remain operational. This means that a consensus protocol can always ensure the correctness of decisions (safety) and, under favourable conditions, also their eventual achievement (liveness).
This result suggests that by relaxing the model's assumptions and adopting more realistic ones, such as partial synchrony or the presence of reliable majorities, the impossibility of reaching a decision can be effectively circumvented. This insight opened the way for practical consensus algorithms, such as Paxos and Raft, which overcome the FLP limitation by introducing timing assumptions or leader-based coordination to ensure termination.
The following section presents the main characteristics of these practical algorithms.

2.3 Practical consensus protocols

In real systems, failures do not occur continuously: there are periods of stability that can be exploited so that a consensus protocol, in addition to ensuring safety, can also eventually terminate.
Practical consensus protocols are therefore based on a set of assumptions about timing, fault models, and the number of failures that can be tolerated. Moreover, several design strategies have been introduced to reduce their communication complexity and make them implementable in practice.

The following paragraphs outline these assumptions and strategies.

Timing and partial synchrony

The timing assumption that characterizes practical consensus protocols is partial synchrony. In this model, message delivery times are finite but unknown (see Chapter 1). Once the system has stabilized, messages are eventually delivered within bounded delays, enabling processes to coordinate and reach an agreement. In practice, protocols implement partial synchrony using *clocks* and *timeouts* to measure time intervals, limit waiting periods for messages, and detect the absence of a response from processes. Consequently, timeouts act as *failure detectors*, enabling the system to identify non-responsive nodes as potentially faulty and make progress rather than wait indefinitely.

Consensus can therefore be achieved during stable periods, provided that a majority of correct nodes remains active.

Fault tolerance and majorities

Using majority is the most intuitive mechanism to guarantee agreement. However, in the presence of faults, it cannot be guaranteed that a majority always exists or that all nodes see the same majority. A message exchange and acknowledgment mechanism is therefore required to ensure that all correct processes share the same view of the system. At the same time, the total number of nodes must be sufficiently large with respect to the number of tolerated failures, to guarantee that a majority of correct processes can always be formed.

There are some results that define these conditions according to the fault model. For instance, under Byzantine faults, a system must include at least $n \geq 3t + 1$ nodes to tolerate t faulty ones [3]. This ensures that any decision quorum of $2t + 1$ nodes overlaps with any other in at least $t + 1$ correct nodes, preventing conflicting decisions. Constructing such quorums requires multiple communication rounds and a quadratic number of messages $O(n^2)$, so that all correct processes can eventually agree on the same majority.

To reduce the number of message exchanges, many Byzantine-tolerant protocols adopt authentication techniques, such as digital signatures, to remove the need for recursive message verification among nodes. This lowers the communication cost at the price of additional local computation for signing and verifying messages.

In crash-tolerant systems, the requirements are weaker: consensus can be achieved as long as a majority of nodes remain active ($n > 2t$), and only two communication phases (request and acknowledgment) are needed for each decision.

In practice, the implementability of a consensus protocol depends on its overall complexity, that is determined by both execution time and the number of exchanged messages. To keep this cost manageable, most practical algorithms adopt simplified assumptions: they consider only crash failures, and rely on mechanisms such as timeouts and message retransmissions to emulate reliable communication channels [3] [14].

Leader-based strategy

To further reduce complexity, many practical protocols adopt a leader-based approach. A designate leader (or proposer) is responsible for proposing a value to the other nodes. In this setting, consensus is reached once a majority of nodes commit the leader's proposal, and nodes no longer need to exchange messages with all the others to verify the consistency of the decision. This approach reduces communication complexity, and simplifies achieving agreement, but introduces the problem of leader failure, which must be handled through leader election mechanisms.

Most practical consensus algorithms adopt this leader-based strategy. Paxos [12] and Practical Byzantine Fault Tolerance (PBFT) [1] are the reference models. Paxos assumes partial synchrony and only crash faults. It defines three logical roles – proposers, acceptors and learners – and guarantees that a value is chosen once a majority of acceptors acknowledge the same proposal. It follows a two-phase protocol (prepare and accept) that ensures safety and eventual termination during stable periods. Moreover, it is the first distributed consensus algorithm that implements a Replicated State Machine.

This model provides the foundation for subsequent consensus protocols, such as Raft, which reorganizes the same principles to make them easier to understand and implement.

Similarly, PBFT extends the leader-based structure of Paxos to tolerate Byzantine faults by introducing authenticated communication and an additional message phase.

Focus of this work

In this thesis, we focus on crash-tolerant consensus algorithms, which are simpler to implement and better suited to our system's requirements (see Chapter 4). The next section presents the Raft algorithm [15] in detail, to illustrate how all the concepts discussed so far are implemented in a practical leader-based protocol. Understanding Raft's internal mechanisms is also fundamental for the comprehension of the custom consensus protocol proposed in Chapter 4.

2.3.1 The Raft algorithm

Raft (Replicated and fault tolerant) is a consensus protocol that implements a Replicated State Machine (see Chapter 1). It was designed based on Paxos with the aim of making it more practical. This was achieved decomposing the logic into three sub-problems: leader election, log replication and safety. This structure makes the Raft algorithm easier to implement while maintaining the same guarantees of correctness and efficiency as Paxos.

Implementation of the Replicated State Machine

Raft implements a Replicated State Machine through a *replicated log*, which ensures that all servers execute the same commands in the same order, producing identical outputs. In particular, each server runs a deterministic state machine and maintains a log that records the sequence of commands derived from client requests.
The interactions between the client, the consensus module, the log and the state machine are illustrated in 2.1, and can be summarized in the following steps:

1. **Client request**
   The client sends a request to the server.

2. **Log replication**
   The server's consensus module receives the request and appends the command (for instance, an assignment like $y \leftarrow 9$) to its local log.
   The module then coordinates with the consensus modules on the other servers to replicate the same log entry and ensure that all logs eventually contain the same sequence of commands.

3. **Command execution**
   The state machine applies the command and produce an output.

4. **Response to the client**
   The output is sent to the client.

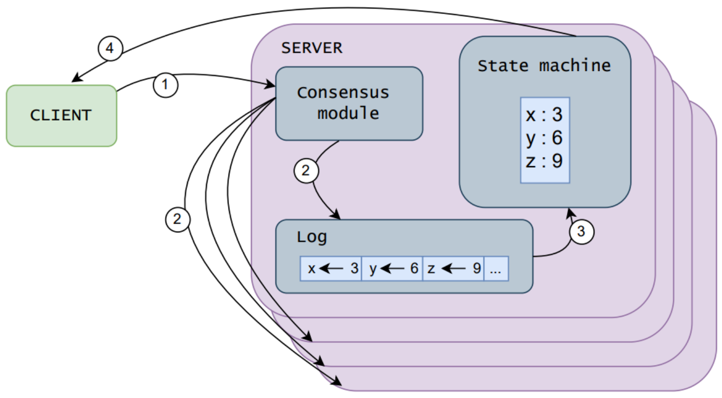

Figure 2.1: Replicated state machine implemented by Raft. Figure inspired by [15].

Thanks to the consensus protocol, replicated logs remain consistent across servers, even if some server fails. In particular, Raft assumes a crash fault model, where servers may stop and later recover. Moreover, messages may be lost and delayed.
As in all practical consensus algorithms, the system can make progress as long as a

majority of servers are operational and can communicate. Under these conditions, Raft ensures both safety and liveness.

What helps Raft to implement this mechanism is the presence of a leader server. In particular, the leader is responsible for receiving client commands, appending them to its log, and replicating them to the other servers in a consistent way.

Overview of the algorithm

In Raft, each server can be in one of three states: follower, candidate, leader. All servers start as followers and remain in this state as long as they receive periodic messages – the *heartbeats* – from a valid leader.

As shown in Figure 2.2, if a follower stops receiving heartbeats from the leader within a given timeout, it transitions to the candidate state and starts an election. It requests votes from other servers – sending RequestVote messages – and, if it receives votes from a majority of them, it becomes the new leader. This is process is known as *leader election*.

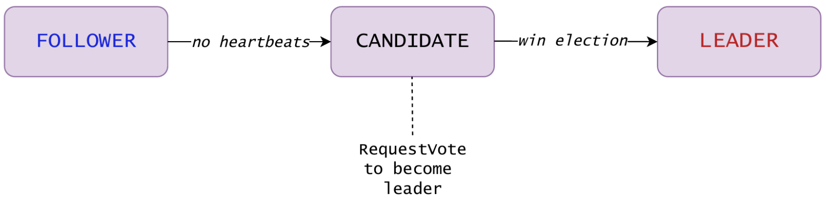

Figure 2.2: State transitions in Raft. Figure inspired by [15].

Once elected, the leader takes full responsibility for client interactions: every client request is sent to the leader, which appends it as a new entry in its log. The leader then replicates this entry to all followers sending AppendEntries messages. These messages also serve as heartbeats to maintain the leadership.

When a log entry has been replicated on a majority of servers, it is marked as committed. The leader applies the corresponding command to its local state machine, returns the result to the client, and notifies the followers that the entry is committed. The followers then apply the same command to their state machines, ensuring that all servers eventually reach the same state. This process is known as *log replication* (Figure 2.3).

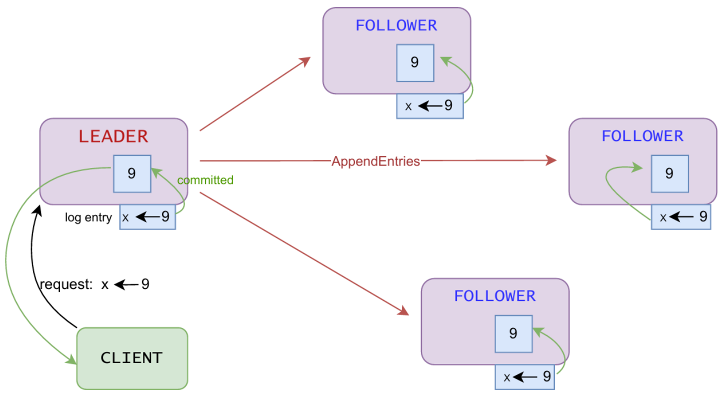

Figure 2.3: Representation of the Raft algorithm logic during log replication. The log entry is appended on a majority of the followers' log, after which it is committed. The leader then applies the entry to its state machine and notifies the followers, which apply it as well.

It is now clear how Raft breaks down the consensus problem into three sub-problems: the leader election, the log replication, and maintaining the safety property. Let's analyze them more in detail.

Leader election

The leader election is characterized by the following notions:

- Election timeout: it is the time a follower waits before starting a new election. It is chosen randomly in the interval $[T, 2T]$ (usually 150–300 ms). This randomization ensures *liveness*: since timeouts are independent, it is highly probable that only one server's timer expires first. That server becomes a candidate and can complete the election before others time out.

- Term: when a follower becomes a candidate, it starts a new *term* and requests votes from the other servers through RequestVote messages. A term is a period of time that begins with an election (when one or more candidates attempt to become leader) and lasts until a new leader is elected. Each term is identified by a monotonically increasing *term number*, stored both in each log entry. Term numbers allow Raft to prevent inconsistencies between logs: if a node receives a message with a smaller term, it recognizes it as outdated and rejects it.

- Uniqueness of the leader: when a follower receives a RequestVote message, it gives its vote if it has not yet voted in the current term. Each follower can vote for only one candidate per term, which ensures a *safety* property: at most one leader can be elected in each term, which prevents log inconsistencies. This is because a leader requires a majority of votes to win, and two different candidates cannot simultaneously obtain overlapping majorities.

- Heartbeat timeout: once a candidate collects votes from a majority of servers, it becomes the leader for that term. The leader immediately begins sending AppendEntries messages to all followers at regular intervals determined by the *heartbeat timeout.* These messages replicate log entries and act as heartbeats to prevent followers from starting new elections.
  New leader elections typically start when a leader crashes or becomes unreachable.

Although randomized timeouts make it very likely that only one follower's timer expires first, simultaneous expirations may occur. In this case, two servers can both start an election for the same term and collect votes from different subsets of followers, resulting in what is called a *split vote.* No leader is elected in that term: since each server votes only once per term, no two candidates can both obtain a majority. After another randomized timeout, a new election takes place and eventually one candidate succeeds in obtaining the majority.

Log replication

Raft uses the AppendEntries procedure to replicate log entries from the leader to its followers. Each log entry is identified by its *index* and the *term* in which it was created.
When sending an AppendEntries message, the leader includes the index and term of the entry that immediately precedes the new ones. This allows the follower to verify log consistency before appending the entries.

A follower accepts the new entries only if it finds in its log an entry with the same index and term indicated by the leader; otherwise, it rejects the message. This consistency check is repeated until a matching entry is found, and then the new entries are accepted. This guarantees that if two logs contain an entry with the same index and term, they are identical up to that point (the Log Matching Property [15]), ensuring *safety.*

When inconsistencies occur, such as after leader crashes or network partitions, the current leader corrects them by overwriting conflicting entries on followers. This mechanism can be illustrated with an example.

Example: handling network partitions

Consider a cluster of five nodes where a partition separates nodes A and B from nodes C, D, and E. Because of the partition, now there are two leader per part with different terms. For instance, node B for term 1 and node C for term 2, elected after the network partition (Figure 2.4).
In this setting leader B can't replicate its log to a majority of nodes, so its log entries remain uncommitted. On the other side, leader C can reach a majority and replicates its log.

When the partition heals, the nodes exchange messages containing their current term numbers. Upon discovering a higher term, the outdated leader B immediately steps down and reverts to the follower state. Any uncommitted entries created during the partition are rolled back, and the followers' logs are aligned with the

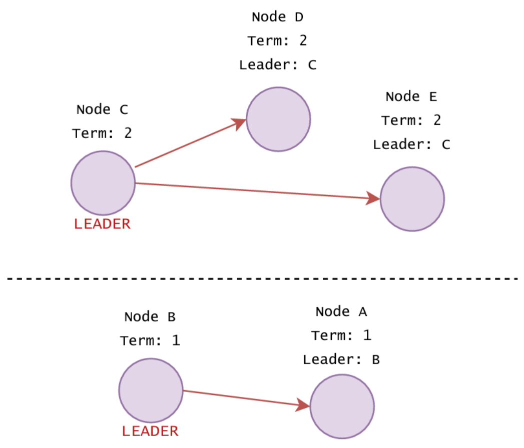

Figure 2.4: Example of a network partition. Node C is elected leader and it communicates with a majority of nodes.

new leader's log through the AppendEntries consistency check.
At the end, all servers converge to the same log.

Safety

Unlike Paxos, Raft explicitly separates the safety aspect of consensus, making it easier to understand and verify. In particular, Raft defines the mechanisms that preserve safety and formalizes them through a set of well-defined properties [15]. A detailed discussion of these properties is omitted here, as the main safety guarantees, such as the uniqueness of the leader and the consistency of replicated logs, have already been illustrated through the description of leader election and log replication.

Chapter 3

Reputation systems

In previous chapters, we introduced the concept of reliability as a fundamental property of distributed systems and discussed how consensus algorithms enable nodes to behave consistently, even when operating in unreliable environments.
To further explore this concept, this chapter introduces reputation systems in distributed environments, where reputation serves as an additional mechanism for assessing and supporting the reliability of the system.

Reputation systems emerged alongside the rise of the Internet as a response to the challenge of building trust between individuals with no prior relationship who interact through digital platforms. These systems collect, aggregate and distribute information about participants' past behaviour, enabling users to decide who to trust and who to interact with. At the same time, it encourages honest behaviour and reduces the overall risk in the online environment.

These mechanisms have become essential in many contexts, including online communities, e-commerce platforms and digital services in general. Due to their applicability to such a wide range of domains, coupled with a lack of standardization, the development of reputation systems remains a complex and active area of research. As well as design considerations, reputation systems present security and privacy challenges, as they must ensure that reputation data remains both reliable and confidential.

Reputation systems are particularly relevant in distributed environments, where there is no central authority to trust and no pre-existing trust between participants. In this context, reputation is used to observe and estimate the reliability of agents, thereby improving the reliability of the overall system.
This chapter provides the general background on reputation systems required to understand the rationale and design choices discussed in Chapter 5. It clarifies why reputation plays a key role in distributed systems and emphasises its relationship with consensus protocols.

3.1 Concepts of reputation and trust

Reputation and trust are the two fundamental concepts on which reputation systems are built. In this section, we adopt the definitions proposed by Jøsang et al. [10].

Reputation is defined as a collective measure of an agent's *trustworthiness*, or how an entity is perceived by others based on its past behaviour. It is obtained by aggregating information from the agents with whom the entity has previously interacted.

Trust, by contrast, is an individual belief. It represents the extent to which one agent is willing to rely on another based on their personal opinion or expectations regarding future behaviour.

While trust is subjective, reputation is based on past experiences (either direct or indirect). In this work, we focus exclusively on reputation, excluding the subjective components of trust.
The term trustworthiness is used in a general sense to describe how reliable an agent appears to others. In our context, it is closely related to the notion of reliability introduced in Chapter 1, as both describe an agent's ability to behave consistently and correctly over time.

Reputation thus represents an estimate of trustworthiness. It is derived from direct observations or feedback from other agents, reflecting the idea that reputation is built progressively over time. The information collected from past interactions must then be translated into interpretable reputation values using specific *reputation functions*. These values can take different forms, such as a numerical score (e.g. a rating from 1 to 10), a discrete value (e.g. a certain number of stars) or a qualitative label (e.g. very good, good or bad). A reputation system defines the mechanisms through which these values are created, managed and exchanged. These mechanisms are discussed in the next section.

3.2 Reputation systems

The basic idea of a reputation system is that agents rate each other after each interaction.
Suppose we want to build a reputation for an agent A: each agent that interacts with A assigns a *rating* after the interaction, that indicates whether the experience was positive or negative: for example, $+1$ for a successful interaction, -1 for a failed one.
All the ratings are then processed by the reputation system, which aggregates them using a reputation function to produce a reputation score. This score provides a value that represents the past behaviour of agent A and can be used by other agents to decide whether or not to interact with A.

According to Swamynathan et al. [17], four main processes can be identified:

1. *Collection* of ratings after each interaction.
2. *Aggregation* of ratings into reputation scores through a reputation function; this is also known as *reputation computation* and also involves updating the scores.
3. *Storage* of reputation scores to ensure they remain available to all agents that need to access them.
4. *Propagation* of the reputation scores to the agents, allowing them to make informed decisions in future interactions.

The specific implementation of these processes defines the architectural model of the reputation system, which can be either centralized or distributed.

In centralized reputation systems, a central authority is responsible for collecting ratings, aggregating them into reputation scores, and storing these values in a global repository. When needed, the central authority provide the reputation scores, or it makes them publicly available. An example is the online marketplace eBay, where the platform itself aggregates ratings from participants to assign a public reputation score to each of them.

In distributed reputation systems, there is no central authority responsible for managing reputation information. Instead, agents can submit their ratings to distributed stores that are accessible to other agents, or they can store their own ratings locally. In both cases, each agent is responsible for collecting available ratings and computing reputation scores locally. Propagation typically occurs on request. Such systems are common in peer-to-peer networks, where participants exchange ratings to estimate each other's reliability.

3.3 The role of reputation in distributed systems

As discussed in Chapter 1, distributed systems operate without central control. Their nodes can be viewed as agents that must collaborate with each other despite not necessarily knowing or trusting one another. This creates an unreliable environment in which agents may fail or act maliciously. In this context, the concept of reputation is particularly useful, as it can be used to estimate the reliability of agents within the system.

In distributed environments, agents that do not know each other often need to collaborate and reach a common decision, which is achieved through consensus protocols. Conversely, reputation mechanisms allow individual agents to make their own decisions, such as whether to trust or interact with another agent, based on collective information. In both cases, the aim is to facilitate reliable decision-making in the absence of trust.

The concepts of consensus and reputation often intersect in the literature. Reputation can be used to support consensus, thereby improving the quality or robustness of the agreement process, an idea that is explored in Chapter 5 of this work. Conversely, consensus mechanisms can be employed to reach an agreement on reputation values themselves, thereby ensuring consistency in how agents perceive the reliability of others.

The ultimate goal of both mechanisms is to ensure the system's reliability. A reputation system contributes to this by estimating the reliability of individual agents, thereby supporting the reliability of the system as a whole.
These ideas are developed further in the following chapters. Chapter 4 presents the design of a consensus protocol tailored to the characteristics of the distributed environment introduced in this work. Chapter 5 then extends this design by incorporating reputation as a supporting mechanism for consensus, showing how the reliability of agents can be used to enhance the reliability of the system itself.

Chapter 4

Consensus Algorithm on a star topology network

This chapter presents the case study that inspired this thesis, which focuses on the study of a consensus algorithm for a distributed system designed to manage critical files. The project was developed in a real-world context with the aim of creating a system that coordinates multiple independent storage services.

A key part of the project involved tailoring the algorithm to the architectural constraints imposed by a star topology. This architecture exhibits hybrid characteristics, combining elements of centralized and distributed approaches. A thorough analysis of the system's behaviour was required to capture these characteristics, which led to the development of a customized consensus protocol.

The chapter begins by introducing the distributed system under study. It then presents the design phase of the proposed consensus algorithm, highlighting the design choices derived from the theoretical results presented in Chapters 1 and 2. It then describes the implementation of the Algorithm, and concludes with the simulation framework used to validate its logic and evaluate its performance in different scenarios.

4.1 Overview of the case study

This case study involves a distributed system intended to manage a file while maintaining the properties of reliability, availability and consistency (as introduced in Chapter 1). The need for such a system arises from the nature of the data contained within the file. As this data is highly sensitive, the primary objective is to preserve its content. Although encryption techniques are employed to protect the content itself, these lie beyond the scope of this thesis. In this study, the file is considered as a whole without examining its internal content. Within this framework, file consistency is understood as coherence among replicas of the same file and serves as an additional safeguard for the content.
Therefore, the main goal of this system is to provide a reliable storage service to its users.

4.1.1 The architecture

The system is based on a client-server architecture developed over a star topology network (Figure 4.1), in which the client acts as the central node communicating with multiple server nodes. This configuration can be considered as a distributed implementation of the client–server model (see Chapter 1).
While the architecture exhibits distributed characteristics, it still relies on a single coordinating client. Similar configurations, in which a central hub connects a set of peripheral nodes, are commonly referred to in the literature as *hub-and-spoke* networks.

It is important to note that the server nodes do not communicate directly with each other, resulting in a centralized communication pattern. Therefore, the system cannot be classified as a hybrid architecture, as it does not exhibit peer-to-peer features.

4.1.2 The distributed system

As mentioned, the primary objective of the system is to preserve a critical file, providing a reliable storage service. To this end, the file is stored on multiple

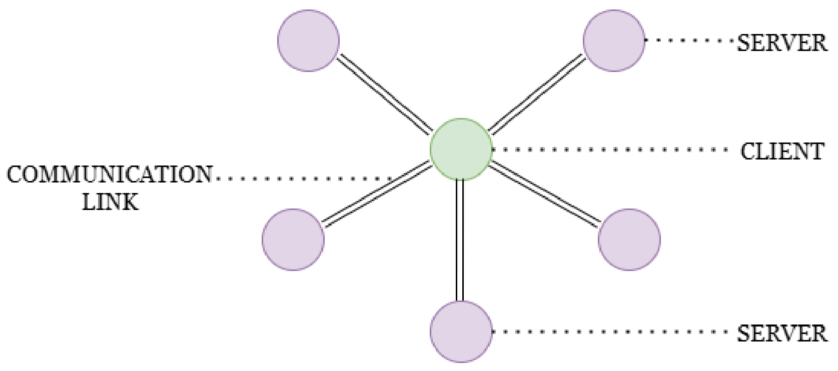

Figure 4.1: Representation of a network with a star topology.

independent storage services. These services are heterogeneous and not part of the same infrastructure, meaning that the system relies on third-party storage providers rather than operating its own storage infrastructure. In this sense, the servers are considered to be externally managed.

To store the file, the client sends the same request to each server node. Three types of operation are supported:

- a `save` request to store the file for the first time;
- an `update` request to fully overwrite the file;
- a `restore` request to retrieve the file from the servers.

By distributing the file across multiple servers, the system prevents data loss. In particular, it adopts data replication as the fundamental technique for increasing reliability and availability, as discussed in Chapter 1.

In many distributed systems, replication is often introduced as a supporting mechanism to enhance overall functionality. In this case study, however, replication is the core design principle: the system relies on full replication, with each server holding a complete, independent copy of the file. As a result, it behaves as a distributed data store.

4.1.3 Motivations

Storing the file across multiple servers directly improves system reliability by eliminating the risk of a single point of failure. Consequently, the system can withstand individual server crashes without affecting the overall service. Reliability is the most important requirement in this system given the valuable nature of the file's contents, as data loss could result in financial damage. This strategy also reduces dependence on a single storage provider, thereby mitigating the risks associated with centralization.

In particular, replication increases fault tolerance and availability, guaranteeing that the file remains accessible even in the event of faults or network issues such as partitions or communication delays.

However, replication is not sufficient to guarantee the system's correct behaviour in the presence of faults. As already discussed in Chapter 1, data replication leads to inconsistency problems. For instance, if an update is acknowledged by some replicas but missed by others due to a crash, the system may end up in an inconsistent state. To address this issue, the system adopts the State Machine Replication (SMR) model (see Section 1.2.1). In this model, all servers behave as deterministic state machines that process the same sequence of operations and therefore remain consistent with one another. In the context of this case study, SMR provides the conceptual framework for coordinating file updates across replicas.
The key challenge is to maintain a common ordering of operations among all nodes. To this end, the system relies on a consensus algorithm that ensures all replicas agree on the same sequence of updates, so that even in the presence of failures, all non-faulty nodes apply identical operations and preserve consistency.

Paxos was among the first algorithms specifically designed to implement SMR and it remains a foundational reference. The algorithm proposed in this work follows the same paradigm, using distributed consensus as its core coordination mechanism. However, due to the specific architecture and assumptions of the system under study, the resulting protocol differs from traditional algorithms in several aspects (see Section 4.3).

4.2 Algorithm design

This work began with an analysis of fundamental results concerning consensus in distributed systems. The FLP impossibility result and the CAP theorem represent the theoretical foundations that define the limits of what can be achieved. In practice, the most influential consensus algorithms are Paxos and Raft, which offer practical solutions to the consensus problem based on realistic assumptions.

Understanding these results was essential in order to evaluate whether any of the existing approaches could meet the specific requirements of our system, and if not, how they should be adapted. The following subsections discuss how these theoretical and practical contributions informed the design choices presented in this work.

4.2.1 FLP-related design choices

As with any consensus algorithm, the design process had to address the limitations imposed by the FLP impossibility result (see Section 2.2). As a workaround, we relaxed the hypothesis relating to the network. Specifically, we moved from an assumption of network asynchrony to an assumption of partial synchrony. The rationale behind this design choice and its implications are discussed below.

Asynchronous communication, where message delays are unbounded, is a realistic assumption and reflects the way our system basically operates. In our context, a delayed message corresponds to a delayed reply[1] from a server. Network asynchrony introduces challenges in distributed systems by creating undecidable states. When a reply fails to arrive, the system cannot determine whether a node has crashed or whether the message is merely delayed in transit. This ambiguity prevents consensus algorithms from making safe progress, an issue that is formally captured by the FLP impossibility result.

[1]The classical client-server model communication follows a request–reply pattern: the client sends a request to each server and expects a reply.

As discussed in 2.3, this limitation can be overcome by introducing timing assumptions. The partial synchrony model, assumes that the system is initially asynchronous, but will eventually stabilize. After an unknown global stabilization time, all messages are delivered within a bounded delay. Our system uses timeouts as a pragmatic approximation of these bounded delays: when a node fails to respond within the designated timeout period, the system can progress by treating this as a failure. This approach effectively distinguishes between transient delays and permanent failures. However, it cannot distinguish between a server that has genuinely crashed and one that is deliberately unresponsive. For instance, a Byzantine node may deliberately withhold its response to disrupt the system. Such behaviour is outside the scope of our fault model.

Another pragmatic response to the FLP impossibility result is the adoption of a leader-based strategy. This approach helps guarantee progress in partially synchronous systems. As discussed in 2.3, Paxos is the reference algorithm. It also relies on timeouts to detect leader failures and trigger new elections, leveraging timing assumptions to make the election process effective.
In addition to leader election, the Paxos algorithm implements a replicated state machine. While leader election addresses liveness by ensuring that one node can eventually drive the protocol and prevent indefinite blocking, the replicated state machine guarantees safety by ensuring that, once a decision has been made, it is irrevocable and applied consistently across all non-faulty replicas. Together, these techniques enable Paxos to overcome the limitations of fully asynchronous systems, establishing it as the foundational algorithm for practical consensus protocols.
The same principles also inform the design of our consensus algorithm, which adopts a leader-based approach and the paradigm of the replicated state machine.

Having discussed the design implications derived from FLP, we will now turn to the practical consequences of another fundamental impossibility result in distributed systems: the CAP theorem (see Section 1.3).

4.2.2 CAP-Theorem oriented design considerations

The CAP theorem is often invoked when designing distributed systems because it provides a useful framework for reasoning about the trade-offs between consistency and availability in the event of network partitions. However, as discussed in 1.3, the theorem relies on precise and restrictive definitions of the three properties involved:

- Consistency refers to linearizability, which requires operations to appear instantaneous and totally ordered.
- Availability means that every request to a non-failing node eventually receives a valid response.
- Partition tolerance implies continuing to operate despite communication failures between nodes.

Real systems may not fully match the strict definitions assumed by the CAP theorem. In practice, they operate under weaker or context-dependent guarantees of consistency and availability. Thus, rather than treating CAP as a rigid classification tool, it is more useful to adopt it to examine the specific behaviour of a system. This helps to identify the system's goals and highlights how its actual properties of consistency and availability diverge from the strict CAP definitions.

The system presented here cannot achieve consistency in the CAP sense due to concrete architectural constraints. Specifically, the server nodes represent independent external storage providers that remain outside the client's direct control. This prevents the system from enforcing the atomic and totally ordered execution of operations across all replicas.

A closer analysis of the system's consistency behaviour reveals monotonic inconsistency. This occurs when one or more replicas temporarily miss a file update due to network partitions or server failures. Consequently, different servers may temporarily store different versions of the file, with some holding outdated copies and others being up to date.

Similarly, the system cannot meet the CAP definition of availability, whereby any non-faulty node must respond successfully. In our case, certain nodes may be temporarily unreachable, or they may return outdated versions of the file due to missed updates. Nevertheless, the system provides a practical form of availability in that clients can access any reachable replica at any time, even in the presence of network partitions or server failures. The trade-off here is that some requests may return stale data. To mitigate this issue, a dedicated consensus protocol is employed to allow clients to reconcile different versions and select the most recent and valid one.

4.2.3 Raft as the reference algorithm

Although Paxos is widely regarded as the foundational consensus algorithm, its complexity and limited applicability have prompted the search for a simpler alternative. The Raft algorithm [15] was introduced by Ongaro and Ousterhout with the specific aim of providing a more accessible foundation for teaching and system building purposes. As discussed in Section 2.3.1, Raft addresses this challenge by decomposing the consensus algorithm logic into three independent sub-problems: leader election, log replication and safety.
A distinctive feature of Raft is its leader-centric design. As the authors emphasize in [15], Raft treats leader election as a fundamental part of its consensus mechanism, delegating most coordination activities to the leader node. Another central element is the reliance on absolute majorities. A node can only be elected leader with the support of a majority of followers, ensuring election safety. Similarly, a log entry is considered committed once it has been replicated on a majority of followers (see 2.3.1 for details). Although these majority-based mechanisms are simple in principle, they are essential to Raft's fault tolerance and will play a central role in the customized protocol developed in the following sections.

Raft was therefore chosen as the reference algorithm for this study because it provides a clear and modular framework for understanding how consensus ensures consistency in replicated state machines. Moreover, its design principles (leader-based coordination, majority agreement and replication) closely align with the requirements of our replication-based architecture. Raft also provided the basis for implementing a practical consensus mechanism adapted to the system under study.

4.3 Design and implementation of the Consensus Algorithm

The previous section outlined the reasoning behind certain algorithmic choices, which are based on fundamental theoretical results. Turning now to the system's peculiar architecture, the Raft algorithm was initially selected and adapted to fit the characteristics of this case study. However, as the design evolved, it became clear that several Raft components were incompatible with the system's architecture. Consequently, the final algorithm diverges substantially from the original Raft protocol. While some conceptual elements, such as the use of a leader and reliance on majority agreement, were inspired by Raft, the overall protocol was redesigned to address the specific challenges posed by the system under study.

4.3.1 System model and assumptions

The distributed system consists of a client C and a set of servers $S = \{S_1, \dots, S_n\}$. The number n of servers is not fixed: the model is kept general as the focus of this work is on the design and implementation of the protocol rather than on fault tolerance analysis, which depends on the replication factor. The only natural constraint is that $n \geq 3$, since inconsistencies could not be addressed with only two replicas.

File model

The object of the protocol is a file F. Each server S_i maintains a local copy of F, represented as a pair (v_i, h_i), where:

- $v_i \in \mathbb{N}$ is the version number
- $h_i = H(F)$ is the SHA-256 hash function[2] of the file content

[2]A hash function is a deterministic function H that maps an input of arbitrary length to a fixed-length output, called *digest*. It satisfies several properties, including collision resistance,

The version number and hash function are used to identify any inconsistencies between replicas.

Operations

The consensus protocol supports the following operations, which are all initiated by the client C:

- save(F): store an initial copy of file F on all servers
- update(F): update file F across all servers and increase the version number
- restore(): retrieve the file from the servers and decide on a consistent version using the consensus protocol

Roles

Client C acts as the leader in the consensus protocol; the servers S_i act as followers.

It is worth noting that, in classical consensus algorithms, the leader is one of the servers within the cluster. In this case, however, the leader coincides with the client, which does not persistently store the file. This distinction reflects the system's specific architecture and motivated the proposal of a new protocol. Moreover, depending on the deployment, the client may be considered either a coordination service or part of the end-user environment.

Communication model

As mentioned, it is assumed that the network is partially synchronous, which means that message delays are finite but unknown.

In practice, communication follows the request-reply paradigm:

meaning that it is computationally infeasible to find two distinct inputs that produce the same digest. SHA-256 is a hash function that produces a 256-bit digest. In this work, it is used to uniquely identify each file replica because, due to its collision resistance, two files with different content will produce different digests, enabling secure and efficient comparison of the replicas.

- the client sends a request to each server and waits for an acknowledgment (`ACK`) within a fixed timeout Δ;
- if the acknowledgment (`ACK`) is not received within the timeout Δ, the server could be considered unavailable.

Servers do not communicate with each other.

We note that, in this protocol, acknowledgements are primarily used as *failure detectors*. Although they are inherently weak, they enable the client to identify when the server is unavailable. Furthermore, acknowledgements only confirm that a message has been received, not that it has been executed correctly. Consequently, they are not useful for achieving consensus or consistency.

Fault model

The fault model assumed in this work only considers crash faults: a server S_i may become unresponsive and stop processing requests. After recovery, it may return an outdated version of the file.
Byzantine behaviours are excluded. This assumption is reasonable since the system operates within a closed group of authorized nodes. However, the threat of an *honest-but-curious* node is still considered relevant: nodes are assumed to correctly follow the protocol, but they may attempt to access sensitive data.
This fault model justifies the use of simple fault-handling mechanisms, eliminating the need for costly Byzantine-resilient protocols.

In this system, the servers are managed externally, meaning their behaviour cannot be verified directly. This limitation has led to the design of a new consensus mechanism, while Chapter 5 suggests improving the protocol by integrating a reputation system that classifies servers based on their reliability over time.

Consensus goal

The consensus protocol enables the client to identify a single, valid version of the file F. This mechanism relies solely on client-side observations and does not involve any direct coordination between servers. A unique version is selected by applying a majority rule to the responses received.

4.3.2 Leader election in a star topology

In the classical Raft model, the leader node receives client requests and coordinates the consensus process by exchanging messages with all the other nodes. However, this assumption is incompatible with a star topology, in which only the central node C communicates with the peripheral nodes $S_1, \ldots, S_n$, without any interaction among the S_i.

Consequently, the leader election mechanism in Raft fails to operate correctly in this setting. When the initial leader becomes unavailable and stops sending heartbeats, one of the followers S_i may initiate a new election by transitioning to the candidate state. Yet, since candidates cannot collect votes from other servers, it becomes impossible to reach a majority and elect a new leader.

To overcome this limitation, the client node C is assigned the role of fixed leader for the entire duration of the protocol. This reflects the structural role of C as the only node with full connectivity to all replicas, as well as its responsibility for issuing `Update(F)` and `Restore()` requests.

4.3.3 Update and Restore: two phases of consensus

The design of the protocol is structured around two distinct phases, each capturing a different notion of consensus. These phases, named Update and Restore, are not executed in a fixed sequence, but rather in response to explicit requests from client C.

During the Update phase, the client sends the new file version to all the servers $S_1, \ldots, S_n$, with the aim of applying the same modification to all the replicas. Since C acts as a fixed leader and communicates directly with each server S_i, it can replicate the operation uniformly. In this phase, consensus refers to the agreement on the operation to be applied, which is similar to the log replication mechanism in Raft.

The following pseudocode illustrates the basic Update phase mechanism. This logic will later be extended to incorporate failure handling strategies.

```
Update Phase

Client C:
   For each server $S_i \in S$:
       send update(F) to $S_i$
   Wait up to $\Delta$ for ACKs
   For each server $S_i$:
       if ACK not received within $\Delta$:
           mark $S_i$ as unavailable
```

However, due to the lack of guarantees regarding the servers' internal behaviour, this phase alone does not ensure global consistency, as it is not possible to directly verify whether all replicas have actually applied the update.

For this reason, the second phase, Restore, is introduced. Here, the objective is to identify the valid version of file F among those returned by the servers rather than to agree on an operation. When C receives copies from each S_i, inconsistencies may arise due to previous faults, delays, or failed updates. The protocol resolves this issue by applying a majority rule: the version returned by a strict majority of servers is selected as the correct one.

This rule is analogous to the majority-based commitment mechanism in Raft, enabling the client to determine a unique outcome even in the presence of partial failures.

The pseudocode below summarizes the Restore phase. This logic will also be extended in the next subsection to address edge cases and inconsistencies.

```
Restore Phase

Client C:
    For each available server $S_i$:
        send restore() request
    Collect responses $(v_i, h_i)$ within $\Delta$
    Group responses by $(v_i, h_i)$
    If a group has size > n/2:
        accept the corresponding version $F_i$
    Else:
        mark operation as failed
```

4.3.4 Handling failures and fallback strategies

The protocol incorporates fallback strategies to enhance fault tolerance in the event of network delays or node failures. These mechanisms apply to both phases of the protocol, using timeouts and limited retries to ensure progress is made.

Update Phase

During the Update Phase, the client C expects an acknowledgement `ACK` from each server S_i within a fixed timeout Δ. If a server fails to respond in time, the request is retried up to a maximum number of times, denoted as *retry_limit*. This increases fault tolerance by mitigating the effects of temporary delays or failures during the propagation of updates.

```
Fallback during Update Phase
```

```

Client C:
    For each server $S_i \in S$:
        retry counter = 0
        While retry counter < retry_limit:
            send update(F) to $S_i$
            wait $\Delta$ for ACK
            if ACK received:
                break
            else:
                retry counter += 1
        If no ACK received after retry_limit:
            mark $S_i$ as unavailable
```

Note that acknowledgements are only used to detect unavailability. Therefore, while this retry policy does not directly influence consensus decisions, it does ensure the broader dissemination of updates.

Restore Phase

In the Restore Phase, retry mechanisms serve an additional purpose. Beside improving fault tolerance, they also increase the likelihood of obtaining a majority of matching responses. This is because additional attempts give temporarily unavailable servers the opportunity to recover and reply, and allow the protocol to collect delayed responses. This is a necessary condition for selecting a valid version of file F.

If no majority is reached on the first attempt, the client:

- retries the request for up to *retry_limit* rounds.
- waits up to Δ for delayed responses.
- updates the set of available servers accordingly.

```
Fallback during Restore Phase

Client C:
   retry counter = 0
   While retry counter < retry_limit:
       send restore() to all available $S_i$
       wait $\Delta$ for responses
       group responses by version $(v_i, h_i)$
       if any group has size > n/2:
           accept that version
           return success
       else:
           retry counter += 1
           update availability status
    If no majority is found:
        returns all retrieved files
```

These strategies allow the protocol to tolerate failures and continue to make progress without becoming blocked. However, some edge cases remain unsolvable using a simple majority, such as responses that are evenly split or total disagreement. A scenario with a lack of majority is illustrated below.

Evenly split scenario

Suppose the client sends a restore() request to 5 servers. The responses are received as follows:

- S_1 returns version (v_1, h_1) within timeout.
- S_2 returns version (v_2, h_2) within timeout.
- S_3 returns version (v_3, h_3) with slight delay.
- S_4 returns version (v_4, h_4) within timeout.

- S_5 does not respond within the timeout and is considered unavailable for this round.

After collecting all responses, the client compares the hashes and observes:

- two servers (S_1, S_3) returned the same version: (v_1, h_1).
- two servers (S_2, S_4) returned the same version: (v_2, h_2).

No version achieves a strict majority. The protocol cannot determine a valid version and must therefore fall back on retry or alternative mechanisms.

Chapter 5 therefore introduces a weighted majority approach, which uses reputation scores to allow progress to be made even when a strict majority cannot be achieved.

4.3.5 Motivations behind the two phases

The protocol's design into two distinct phases reflects the partially observable nature of the system. The main difference between the Update Phase and the Restore Phase is that the latter is the only phase that enables consistency to be evaluated.

At first glance, it might seem reasonable to apply a majority rule during the update phase: if a majority of servers acknowledge the update, one might assume that the system has reached an agreement. However, this assumption is invalid in our model. Acknowledgements merely indicate that the message has been received, not that the update has been applied. Furthermore, the set of responsive servers may change between the two phases due to crashes or network delays. Consequently, a majority of acknowledgements in the update phase does not guarantee that the same majority will participate in the restore phase, nor that the system has reached a consistent state.

Illustrative example

Suppose the client sends an `update(F)` request with version $v = 3$ to 5 servers, and receives acknowledgments from 3 of them within the timeout:

- S_1, S_2, and S_3 acknowledge the update - majority seemingly reached.
- S_4 and S_5 do not respond - marked as unavailable.

Suppose that later a `restore`() request is sent, and the responses are as follows:

- S_1: no response (crash failure).
- S_2: no response (partitioned).
- S_3: responds after the timeout (considered unavailable for this round).
- S_4: responds with version $v = 2$.
- S_5: responds with version $v = 2$.

Therefore, a majority among the available responses supports version $v = 2$, which is selected as the consistent version.

This example highlights that even if a majority acknowledged version $v = 3$ during the Update Phase, the servers involved in that majority may later become unavailable. This confirms that consistency is enforced only at the level of the values observed by the client during Restore Phase.

4.4 Simulations and results

This section provides a simulation-based evaluation of the consensus protocol outlined in the preceding sections. Simulations were crucial in validating the logic of the protocol, observing its behaviour under various failure conditions, and quantifying its performance and reliability using selected metrics.

4.4.1 Preliminary observations

Initial tests were conducted using a basic implementation of the Raft algorithm to evaluate its compatibility with the star topology. However, the leader election phase failed to reach convergence. This observation prompted the development of a custom consensus protocol.

The simulations presented in this section refer exclusively to the final version of the implementation, which incorporates mechanisms typical of real-world distributed systems. Earlier simulation attempts, which were used to explore and validate the basic logic of the protocol, are not reported.

4.4.2 Simulation setup

The simulation environment is made up of a configurable number of server nodes, one client node and one managed file. The client is responsible for carrying out all main operations, including those relating to the consensus protocol. It is assumed that the server nodes can perform basic storage service operations.

The simulation framework was developed in Python with a modular structure (see the repository `distributed-consensus-star-topology` [3]). Our experiments were designed to evaluate the protocol behaviour under different parameter settings. Since server failures are generated according to predefined probabilities, the simulations are probabilistic and each configuration was tested through 100 simulation runs.

Each simulation run consists of three main phases:

- Initial distribution: the file is initially broadcast to the servers.
- Update Phase: consensus-based propagation of a sequence of updates.
- Restore Phase: consensus-based retrieval and validation of file consistency using a majority rule.

To test the protocol under realistic conditions, several controlled mechanisms are employed:

- Temporal spacing of updates: the client performs multiple file updates with *random* time intervals between each one, in order to simulate the behaviour of an asynchronous system.

[3]`https://github.com/ilariapalumbo/distributed-consensus-star-topology.git`

- Fault injection: according to a *configurable probability*, servers may randomly fail at each update or retrieval attempt. This simulates node faults, including crashes.

- Timeout and retry configuration: in the consensus module, update operations wait for ACKs with a configurable *timeout* and retry each unresponsive server up to a configurable maximum (*retry_limit*). Each attempt is spaced by a configurable interval of time (*retry_period*).

- Tracking of unresponsive and unavailable servers: the consensus protocol marks servers that do not respond with ACKs as *unresponsive.* If they continue to fail, they are eventually marked as *unavailable.*

- Recovery mechanisms: the consensus protocol attempts to re-establish communication with temporarily unavailable servers a maximum number of times, with each attempt being spaced by a configurable interval of time.

- Weighted fallback: in the consensus module, if no absolute majority is reached among the responding servers, the *weighted fallback* strategy is applied. This selects the version supported by the highest cumulative weight.

The weighted fallback strategy was introduced during the implementation phase, given the need to terminate the protocol. The complete implementation of the consensus mechanism, including the weighted fallback logic, is provided in Appendix A.

To better interpret the results, note that update failures are simulated at two levels. At the server level, each update may fail with a given probability to model crashes or internal errors. In such cases, the server does not send an acknowledgement. At the client level, failures are induced by timeouts. If an acknowledgement is delayed or lost due to network latency or partitioning, the server is marked as unresponsive, regardless of whether the update was successfully applied. This distinction allows the simulation to model both functional and communication-related failures.

4.4.3 Evaluation metrics

To evaluate the behaviour of the proposed protocol, a simulation framework was implemented in which files are identified by a version number that increments with each update. In a real system, each file would be uniquely identified by the hash value of its content. However, in the simulation, the hash value and the version number are updated together, so the version index alone is sufficient to track consistency across replicas.

A modelling choice made for these simulations concerns server reliability. Each server is characterized by a failure probability, which abstracts the likelihood of both update and restore failures. In practice, update and restore failures may exhibit different probabilities (for example, write operations are generally more error-prone due to their higher complexity), but for the purposes of this evaluation, a unique failure probability is assumed. This simplification provides a consistent failure model and allows for a more direct assessment of the protocol's robustness.

Failure probabilities are assigned in line with the weights associated with the servers: a higher weight typically reflects a more reliable server and is therefore coupled with a lower failure probability. Nevertheless, even reliable servers can experience network-related issues, such as latency or temporary unavailability. To capture this, each server is also associated with a recovery delay representing the time after failure at which point it can become operational again. This leads to the following consistent strategy: if a server is reliable but subject to latency, it is modelled with a low failure probability and long recovery time. Meanwhile, the protocol parameters are adjusted with higher retry limits and longer retry periods to allow sufficient time for the network to recover.

Within this setup, the evaluation focuses on the protocol's ability to return the latest version of the file during the restore process. Consequently, the *restore accuracy rate*, defined as the proportion of simulations in which the restored file is the latest updated version, is the only significant metric adopted.

The variation in restore accuracy with respect to different protocol and server parameters is reported in tabular form and analysed further in the following section.

4.4.4 Results and observations

The restore accuracy was evaluated across eight different scenarios. Some of these were designed to test the effect of specific parameters under varying conditions, while the others aimed to assess the protocol's overall resilience in adverse environments.

- **Stress_test** represents the most adverse scenario, combining high failure probabilities, long recovery delays, and strict protocol parameters (*retry_limit = 1*, *retry_period = 5 ms* and *ack_timeout = 2 ms*). It serves as a negative baseline, enabling clear observation of the improvements achieved in more permissive or better-balanced scenarios.
- **Higher_retry** is a variation of the previous scenario, in which the effects of a more permissive protocol (with extended retries and long timeouts) are tested, while keeping the servers unchanged. The aim is to verify whether the retry logic can improve accuracy even under very unfavourable basic conditions.
- **Low_retry_High_failure** represents an adverse but more realistic condition than the Stress_test. The aim is to evaluate whether the protocol can maintain an acceptable level of restore accuracy in the event of frequent server failures and limited retry opportunities.
- **High_retry_Medium_failure** is a realistic and balanced configuration, combining moderate failure rates with generous retry settings and moderate recovery delays. The aim is to verify whether the protocol can guarantee good levels of accuracy under average conditions.
- **Reliable** explores a favourable setting with low failure probabilities (between 0.05 and 0.1), quick recovery times and a fair distribution of weights. It establishes an upper bound for protocol performance under ideal conditions.
- **Low_retry_Low_failure** intentionally combines good server behavior with minimal protocol tolerance. The objective is to verify whether the system could still be compromised by restrictive operating conditions.

- **High_retry_High_latency** isolates the effect of delayed server recovery. Despite the failure probabilities being low to moderate, all servers have extended recovery windows. This tests whether an increased retry budget can mitigate the effects of high latency.

- **Weight_fallback_test** is designed to simulate situations where the numerical majority may not be reachable, forcing the system to rely on the fallback strategy based on server weights. The aim is to observe whether a highly weighted and more reliable server can drive the decision in the absence of a numerical majority.

The table below shows the configuration of each scenario and its corresponding accuracy rate.

The results clearly demonstrate the relationship between protocol parameters and restore accuracy. As expected, the Stress_test scenario yields the lowest accuracy 0.46, confirming that strict timeouts and retry limits severely impact system performance when failure probabilities are high and recovery times are long. There is a slight improvement in the Higher_retry configuration 0.55, which uses the same servers as the Stress_test configuration but increases the retry budget and timeout values. This shows that increasing retry opportunities can partially offset the effects of poor network conditions, even if the underlying infrastructure is unreliable.

Further improvement in performance is seen in the Low_retry_High_failure scenario, which achieves an accuracy of 0.82. Here, the failure rates and recovery times are lower than in previous cases. Despite the protocol remaining relatively strict, the system benefits from a more favourable environment. The High_retry_Medium_failure scenario achieves an accuracy of 0.94. This shows that, under more realistic conditions involving moderate server reliability and more permissive protocol parameters, a balanced configuration can result in near-optimal behaviour.

Both Reliable and Low_retry_Low_failure performed extremely well, achieving accuracy values of 0.96 and 0.97, respectively. Interestingly, the latter achieves slightly better results despite having much stricter protocol settings. This suggests that severe protocol constraints do not significantly hinder the system's ability to maintain consistency when the infrastructure is highly reliable.

The High_retry_High_latency configuration confirms that retry mechanisms remain effective in the event of slow server recovery. With a restore accuracy of 0.90, this scenario shows that latency alone does not significantly affect system behaviour, provided the protocol allows enough time for the servers to become responsive again.

Finally, the Weight_fallback_test demonstrates the effectiveness of the weighted fallback strategy, achieving an accuracy of 0.89. Although two of the three servers are unreliable and could form a numerical majority in traditional quorum systems, their failures often prevent that majority from forming. In such cases, the system

Table 4.1: Restore accuracy under different configurations

Config	**r__limit**	**r__period (ms)**	**ack__to (ms)**	**f__prob**	**rec__delays (min–max)**	**weights**	**accuracy**
`Stress_test`	1	5	2	[0.4, 0.5, 0.6]	(30,50), (40,60), (50,70)	[10, 5, 2]	0.46
`Higher_retry`	5	20	10	[0.4, 0.5, 0.6]	(30,50), (40,60), (50,70)	[10, 5, 2]	0.55
`Low_retry_High_failure`	3	10	5	[0.2, 0.3, 0.5]	(15,25), (20,30), (25,35)	[10, 7, 2]	0.82
`High_retry_Medium_failure`	5	20	10	[0.1, 0.2, 0.3]	(5,15), (15,25), (20,30)	[10, 7, 2]	0.94
`Reliable`	3	10	5	[0.05, 0.1, 0.1]	(5,15), (8,18), (10,20)	[10, 9, 8]	0.96
`Low_retry_Low_failure`	1	5	2	[0.05, 0.1, 0.1]	(10,25), (12,30), (15,35)	[10, 9, 8]	0.97
`High_retry_High_latency`	6	20	12	[0.1, 0.2, 0.3]	(30,60), (25,55), (20,50)	[10, 7, 5]	0.90
`Weight_fallback_test`	3	10	6	[0.1, 0.5, 0.5]	(10,25), (10,20), (10,20)	[10, 3, 2]	0.89

relies on the weight of the more reliable server to reach an agreement, enabling the protocol to maintain consistency in a significant proportion of runs.

In conclusion, the role of weights is examined in more detail. In the context of the simulations presented, weights represent the degree of trust assigned to each server. When server behaviour is unpredictable and uncertainty must be managed, reputation offers a more realistic interpretation of this mechanism. From this perspective, the weight of each server is treated as a reputation score, which is dynamically determined by its performance when executing the consensus protocol. The next chapter will present the reputation system that computes and updates these scores and explain how it is integrated into the consensus mechanism.

Chapter 5

Reputation-based extension of the consensus protocol

This chapter explores the relationship between reputation and consensus, investigating how reputation can inform the decision-making process within the consensus protocol outlined in Chapter 4. The fundamental notions of reputation systems on which this chapter builds were introduced in Chapter 3.

Having presented the reasons for introducing a reputation system, we design it as a separate component and subsequently integrate it into the protocol. Therefore, the proposed solution is an extension: while the consensus logic remains valid and operational without reputation, the additional reputation layer aims to improve the quality of decisions in scenarios where the baseline majority rule may fail.

5.1 Motivations and objectives

The introduction of a reputation system is intended to enhance the consensus protocol under several aspects.

Firstly, it aims to strengthen the guarantee of reaching consensus. The baseline protocol relies on a simple majority rule for decision-making and may therefore fail when a majority is absent or tied. In practice, however, the consensus process must always reach a decision. A reputation system addresses this limitation by ranking servers according to their observed reliability and enabling their responses to be weighted accordingly. Thus, the reputation system can break ties in favour of reliable servers, enabling the protocol to reach a decision and ensure liveness.

Secondly, in a setting where servers are considered untrusted agents, they must be monitored to identify unreliable ones. Reputation provides an effective tool for this purpose, enabling the system to protect itself against misbehaviour. Although the underlying consensus protocol is only designed to tolerate faults caused by crashes, integrating reputation enables the system to mitigate a broader class of unreliable behaviours, such as intermittent responsiveness, persistent delays and repeated stale replies.

Beyond the functional aspects, reputation enhances the system's trustworthiness, which is understood as the level of confidence that users place in its operation. Users may question the reliability of a system that relies on external servers. However, by incorporating reputation, the protocol increases confidence in the outcome of consensus decisions, since reliable servers exert greater influence.

Finally, reputation contributes to the scalability of the distributed system. In a setting where the client alone coordinates the consensus process, adding more servers increases the processing overhead of the responses. However, expanding the set of servers improves robustness, as a greater number of replicas increases the likelihood of recovering the correct version in the event of a failure. Reputation helps balance these effects by selecting the most reliable servers, thereby reducing overhead while enabling the system to scale.

The main objective of this study was to extend the baseline consensus protocol with a reputation-based component. This was broken down into three objectives:

1. Design of a reputation system.
   The first objective was to examine the reputation models identified in the literature and select one that was suitable for our context. This analysis

resulted in the adoption of a distributed, voting-based approach. Based on this, we designed a reputation system in which clients evaluate servers based on their responses during the consensus protocol, using these evaluations to build a reputation value that evolves over time.

2. Ledger-free management of reputation.
   A second objective was to determine how reputation values should be stored. In addition to the exclusion of a central authority, the idea of distributing reputation storage across external and potentially untrusted servers was also deemed unfeasible. The adopted solution avoids any form of global storage (whether centralized or distributed) and manages reputation entirely locally by the client instead.

3. Integration of reputation into the consensus process.
   The final objective was to determine how reputation values could influence the consensus protocol's decision-making process. The aim was to expand upon the baseline majority rule, ensuring that decisions were not solely dependent on the number of servers supporting a given version, but also on their assessed reliability. To this end, reputation scores were used as weights in the Restore phase, enabling the protocol to prioritize the responses of more reliable servers.

5.2 The reputation system

This section introduces the proposed reputation system. The first subsection, Reputation model, defines the operating environment and addresses the first two objectives presented above. The subsequent subsections, Reputation Computation and Reputation Propagation, correspond to the fundamental components of any reputation system, describing how reputation is derived and exchanged among agents.

We point out that the current design does not address the risks typically associated with reputation systems, such as security and privacy threats. Nevertheless, we acknowledge that addressing these issues during the design process is crucial to achieving a robust and reliable reputation system. For this reason, in Chapter 6 we discuss the limitations of this work and we outline directions for future work.

5.2.1 The reputation model

The first step in designing a reputation system is to define its operating environment. This involves identifying the agents involved, clarifying their roles and specifying the information available for constructing reputation. It also involves setting communication constraints on how this information can be exchanged.
These basic elements determine how reputation is generated and maintained across the system. The mechanisms through which reputation values are transmitted are addressed separately in 5.2.3.

As outlined in Gurtler and Goldberg's Systematization of Knowledge [8], a broad class of reputation systems distinguishes agents between *voters*, who provide evaluations, and *votees*, who are evaluated. Depending on the context, additional roles may also be defined. This categorization has been adopted in our setting. Clients are naturally modelled as voters, since they interact directly with servers. Meanwhile, untrusted servers act as votees, whose behaviour must be evaluated.

To introduce reputation into our system, we need to expand the underlying network. In the baseline architecture introduced in Chapter 2, a single client coordinates a set of n servers. Here, we assume an expanded setting with a set of $N > n$ servers $\{S_1, \ldots, S_N\}$ and multiple clients $\{C_1, \ldots, C_M\}$.
This expansion is motivated by the intrinsic meaning of reputation, which is understood as an estimate of an agent's reliability based on direct experience, feedback from others, or a combination of the two (see Chapter 3). By including more agents, more information is available in order to build a more accurate estimation of reputation.

To avoid ambiguity, we distinguish between three related notions about reputation:

- a *rating* is the evaluation based on a single client–server interaction,
- a *reputation score* is an aggregated value obtained by combining multiple ratings, which provides a more informative measure of reliability,

- the term *reputation value* is used more generally to denote reputation-related information, without committing to a specific level of granularity (e.g. single interaction vs. aggregated information).

Formally, let each interaction k between a client C_j and a server S_i during an execution of the consensus protocol produce a tuple of observable metrics

$$o_{i,j}^k = (a_{i,j}^k, t_{i,j}^k, opt), \tag{5.1}$$

where:

- $a_{i,j}^k \in \{0{,}1\}$ represents the availability of server S_i in the k-th interaction with client C_j; it is 1 if the server responded, 0 otherwise.
- $t_{i,j}^k \in \mathbb{R}^+$ represents the response time of S_i in the k-th interaction with client C_j.
- optional metrics may be considered depending on the objectives of the reputation system.

These observations are mapped into ratings through a dedicated function (introduced in 5.2.2), and progressively aggregated into reputation scores (see section 5.2.2 for details). In this work, we therefore assume that reputation ratings are outcome-based, and are not intended as subjective opinions of the agents. This makes them verifiable.

In this system, each client must assign a reputation score to every server S_i with which they interact during the execution of the consensus protocol. As mentioned, direct experience is not always sufficient; for example, when a client must decide whether to interact with a new server or when there have not been enough direct interactions to build reliable reputation scores. To address this limitation, clients can request reputation values from others. This requester-witness communication allows each client to estimate the reliability of servers with which it has not yet interacted with, and to select a subset of n servers for each execution of the consensus protocol:

$$S \subset \{S_1, \ldots, S_N\} \quad \text{with} \quad |S| = n < N,$$

which we denote as the *active set*.

Within this extended framework, clients can dynamically assume different roles:

- *voter*: a client that evaluates servers through direct interactions;
- *requester*: a client that queries other clients for reputation values;
- *witness*: a client that provides *feedback* in response to such requests.

Within this framework, producing a rating after an interaction is considered a *voting operation*.

According to the classification proposed by Gurtler and Goldberg [8], our model falls within the category of voting-based reputation systems. Moreover, since no central authority is responsible for managing reputation, the system is distributed. Each client derives scores from its own interactions and can request more information from other participants when necessary. Reputation is therefore voter-owned, as each client locally stores its own scores locally rather than relying on global values accessible to all. In this sense, the management of the reputation is ledger-free.

Several consensus protocols in the literature rely on notions of reputation or trust. Many of these approaches are built on blockchain[1] technology. For instance, Proof of Trust [20] integrates the notion of trust into a blockchain consensus algorithm, whereby nodes with higher trust values are more likely to be selected as block validators. Similarly, Proof of Reputation [5] uses reputation rather than computational power or stake as the scarce resource recorded on the ledger. In both cases, reputation is stored and verified globally through the distributed ledger. This differs from our design, in which reputation is maintained locally by the client and not recorded in any shared structure.

The decision to design a stand-alone reputation system before integrating it into the consensus protocol reflects the requirement to avoid any form of ledger. The reputation system manages reputation values, and the resulting scores are subsequently exploited by the consensus mechanism.

[1]A blockchain is a specific type of Distributed Ledger Technology.

5.2.2 Reputation computation

The central component of any reputation system is the computation of reputation. To this end, a reputation function must be defined to provide a mechanism that aggregates the available inputs into an interpretable value. These inputs may consist of a sequence of ratings produced by a single voter or a set of feedback provided by multiple voters. The outcome of the aggregation must then be mapped onto a reputation score - a stable measure that can be used for decision-making purposes. In the reputation system presented here, the computation of reputation, including the definition of a suitable reputation function, is developed across three levels.

1. From observations to ratings

As discussed in the previous section, each interaction between a voter and a votee produces a set of observations that are converted into a numerical rating representing the voter's evaluation of the interaction. Ratings are assumed to be normalized so that they can be compared and suitably aggregated in later stages of the reputation computation process.
Let $\mathcal{O}$ denote the space of the observations. A *rating function* is defined as a mapping

$$\varphi : \mathcal{O} \to [0{,}1],$$

which transforms an observation into a rating in the unit interval. This corresponds to the *voting operation* of the system.

In the specific setting of this work, let us assume that an observation is expressed as a tuple

$$o_{i,j}^k = (a_{i,j}^k, t_{i,j}^k),$$

where $a_{i,j}^k$ and $t_{i,j}^k$ are the two metrics defined in (5.1). A possible choice of the rating function is a linear mapping:

$$\varphi(o_{i,j}^k) = \begin{cases} 0 & \text{if } a_{i,j}^k = 0, \\ \alpha \cdot a_{i,j}^k + \beta \cdot \left(1 - \dfrac{t_{i,j}^k}{T_{\max}}\right) & \text{if } a_{i,j}^k = 1, \end{cases}$$

where $\alpha, \beta \geq 0$ are weights such that $\alpha + \beta = 1$, reflecting the relative importance assigned to the two metrics. The parameter $T_{\max}$ denotes the maximum response time tolerated within an execution of the consensus protocol, determined by the configuration of timeouts and retry intervals.

2. From ratings to reputation scores

While a rating captures the outcome of a single interaction, a reputation score reflects a voter's cumulative assessment of a votee, obtained by aggregating ratings over time.
Let $R_i^{(j)}(k)$ denote the reputation score of the server S_i maintained by the client C_j after k interactions. Given the new rating $\varphi(o_{i,j}^{k+1})$, the reputation is updated according to the recursive rule

$$R_i^{(j)}(k+1) = (1-\lambda) \cdot R_i^{(j)}(k) + \lambda \cdot \varphi(o_{i,j}^{k+1}) \tag{5.2}$$

where $\lambda \in (0{,}1]$ is a parameter that controls the balance between memory and new evidence. In this way, reputation scores evolve dynamically, taking into account recent behaviour while retaining information from previous interactions.

3. From feedback to estimates

As previously mentioned, feedback can be used when a client has had insufficient interactions to build a reliable reputation score. In our model, however, once the consensus protocol is running, interactions with the servers are continuously collected and scores are updated accordingly. For simplicity, we therefore consider the beginning of an execution of the protocol - when a client has never interacted with a server - to be the only situation in which feedback is requested.
Let $\mathcal{W}_i^{(j)}$ denote the set of witnesses from whom $C^{(j)}$ requests information about S_i (how this set is obtained is discussed in 5.3.2). The estimated reputation is obtained by aggregating the feedback received:

$$\tilde{R}_i^{(j)} = \mathit{Aggregate}\Big(\{R_i^{(w)} \mid w \in \mathcal{W}_i^{(j)}\}\Big), \tag{5.3}$$

where *Aggregate* denotes an operator that combines values provided by multiple witnesses, such as an arithmetic mean or a weighted average. The definition of this function depends heavily on the context: it depends on the form of the reputation scores and must produce a coherent and meaningful value.

The resulting estimate $\tilde{R}_i^{(j)}$ acts as the initial reputation score of server S_i, taking the place of $R_i^{(j)}$ in the update rule 5.2. Once the protocol starts and interactions with S_i are collected, $\tilde{R}_i^{(j)}$ is updated through the mapping $\varphi(o_{i,j}^k)$, and thereby evolves into the reputation score $R_i^{(j)}$.

Note. The Aggregate operator combines reputation scores directly, rather than raw ratings. This reflects the fact that clients continuously update their local reputation values for each server, meaning that the feedback they provide is already expressed as a reputation score.

Although the direct-update mechanism (see equation 5.2) and the feedback-aggregation mechanism (see equation 5.3) are described separately, both are instances of the general notion of a reputation function, as they both aggregate different forms of input into a reputation score. This distinction stems from the consensus protocol's structure, which is based on direct client–server interactions. The reputation system was therefore designed to reflect this setting: direct experience constitutes the core input, while the aggregation of feedback from other clients is an extension introduced by the reputation mechanism. This layered approach was inspired by the Beta Reputation System (see [9]), which explicitly introduces mechanisms for combining ratings from multiple sources.

5.2.3 Reputation propagation

The propagation of reputation values is a necessary component of any reputation system. In centralized settings, this task is handled by the authority that collects and distributes scores. In decentralized systems, however, this task is carried out directly by the agents and must therefore be defined according to the context.
In the proposed model, propagation occurs when a requester queries a set of witnesses for their reputation scores on a certain server.

Therefore, the propagation mechanism is closely related to the Aggregation

function 5.3.
Each witness $w \in \mathcal{W}_i^{(j)}$ holds a local score $R_i^{(w)} \in [0,1]$ and sends it to the requester $C^{(j)}$ which collects feedback and computes an aggregated value:

$$\tilde{R}_i^{(j)} = \frac{1}{|\mathcal{W}_i^{(j)}|} \sum_{w \in \mathcal{W}_i^{(j)}} R_i^{(w)}. \tag{5.4}$$

The arithmetic mean is a possible choice that preserves comparability with ratings derived from direct experience.

5.3 Integration of reputation into the consensus process

As stated in the objectives of this work, the decision-making process depends on reputation, assigning a weight equal to a server's reputation to each server. This section examines the first adopted strategy – a weighted decision – and highlights its limitations. It also discusses another instance in which reputation values are employed in the consensus protocol: the server selection phase.

5.3.1 Weighted decision in the Restore phase

Let $S^{(j)}$ be the active set of servers associated with client C_j in a given consensus round, i.e., one complete execution of the protocol from initiation to agreement on a file version. During the *Restore* phase, each server $S_i \in S^{(j)}$ returns a version identified by a hash value h_i. Client C_j then partitions the responses by hash value, as required by the consensus protocol (see Chapter 4): each distinct hash value h defines a group of servers that returned the same version. The total weight of a group is the sum of the reputation scores of its servers:

$$V_j^k(h) = \sum_{\substack{S_i \in S^{(j)} \\ h_i = h}} R_i^{(j)}(k),$$

where k denotes the index of the current client–server interaction within the consensus round. Reputation values are updated after every interaction, so $V_j^k(h)$

reflects the most recent scores available at that point.

The chosen version is the one with maximum total weight:

$$h^* = \arg\max_h V_j^k(h).$$

This approach is inspired by weighted voting [6]: the response with the largest weight is chosen; in this case, the responses are the file versions returned by the servers. This simple strategy serves two purposes: it resolves ties and situations where there is no majority without resorting to random selection, and it quantifies the level of support for a given version based on the reliability of the supporting servers (the more reliable the servers, the larger the weight).

Its limitations are clear: a single highly weighted server can dominate a larger group of moderately reliable servers, and certain weight distributions can influence the outcome. The goal of introducing reputation is not to outperform the absolute majority, but rather to enhance liveness and decision quality in situations where a majority is absent, tied or partially observable due to crashes, delays or partitions. The proposed strategy does not guarantee that the chosen version is correct. Rather, it provides additional information with which to assess it, and it motivates the search for more robust, correctness-oriented strategies, which will be the focus of future work.

5.3.2 Server selection

At the beginning of each consensus round, a client C_j selects an *active set* of servers[2]

$$S^{(j)} \subset \{S_1, \ldots, S_N\}, \quad |S^{(j)}| = n < N,$$

based on current reputation scores. A simple policy is to choose the top-n servers according to $R_i^{(j)}(k)$ or $\tilde{R}_i^{(j)}$.

This step highlights a common issue in distributed reputation management: *witness selection.* In this system, clients maintain reputation scores locally and

[2]The active set remains fixed throughout the round.

query witnesses when they lack information about a server. The open question then becomes which witnesses to query, that is, which agents hold useful information about the missing servers. We introduce this issue merely to demonstrate its relevance in our setting, without attempting to solve it. Several approaches have been proposed in the literature, such as referral-based mechanisms (Yu and Singh, cited in [19]), but their integration lies outside the scope of this work. In our system this arises in two scenarios:

1. a client already in the system wishes to replace one or more servers. Client C_j maintains local scores for the current active servers in $S^{(j)}$ and needs information only for servers $S_i \notin S^{(j)}$. It should therefore query witnesses who interacted with S_i to obtain the estimates $\tilde{R}_i^{(j)}$;

2. a new client joins the system and acts as a requester. It must select a set of witnesses that provides sufficient coverage of $S_1, \ldots, S_N$, in order to build an overview of the system and then choose the best servers to interact with.

Once the witness sets $\mathcal{W}_i^{(j)}$ have been identified, the estimates $\tilde{R}_i^{(j)}$ can be computed via (5.3).

This server-selection policy is also the base the improve system scalability by focusing on the most reliable servers.

Chapter 6

Limitations and future work

This short chapter outlines the main limitations of this work, primarily concerning the reputation system introduced in the previous chapter. The study of reputation models revealed several key aspects essential for designing a reliable and robust reputation mechanism, particularly with regard to security and privacy. As acknowledged in Chapter 5, these aspects fall outside the scope of the present design. The limitations reported here are therefore presented to indicate directions for future work.

- **Security of propagation**
  The security of communication between the requester and the witnesses was not explicitly addressed in the design of the reputation system. This is a limitation since the propagation mechanism naturally exposes reputation scores to potential attacks. In practice, the system should guarantee the integrity of scores during transmission to prevent tampering, i.e. the alteration of values in transit by malicious parties.

- **Privacy of propagation**
  The propagation phase also raises privacy concerns. Collusion among agents may reveal information about individual evaluations or enable different actions to be linked together. As discussed in the Systematization of Knowledge [8], many existing systems are highly linkable, meaning that a user's actions - such as the votes they cast or receive - can be associated with the same subject.

This enables the reconstruction of behavioural patterns. In our setting, the main concern is witnesses: the system must guarantee that votes cannot be linked to voters, ensuring that the requester cannot associate scores with specific witnesses.

- **Absence of a threat model**
  Another limitation is the lack of a formal threat model. As noted by Swamynathan, Almeroth and Zhao in The design of a reliable reputation system [17], a reliable reputation mechanism must consider threats in all reputation management processes, including collection, aggregation, storage and communication. Therefore, a threat model tailored to our setting would be essential in order to identify possible attacks and design appropriate mitigations.

Future work should address the above-identified security and privacy limitations. Encryption and authenticated channels could provide a basic level of protection for transmitted scores. Privacy-preserving aggregation techniques, such as those based on secure multi-party computation, could be explored in order to compute aggregated values while ensuring that votes cannot be linked to voters. Most importantly, a formal threat model should be defined in order to classify possible attacks and establish a robust reputation system.

Beyond these areas, further investigation is needed to design aggregation strategies that are more robust than the weighted decision illustrated in Section 5.3.

Conclusion

In this thesis, we have addressed the problem of coordinating multiple independent storage services in order to manage critical files reliably. To exploit the properties of distributed systems while maintaining file consistency, a consensus algorithm has been designed that meets the specific architectural constraints of the system. Furthermore, a reputation-based extension has been provided in order to guarantee the termination of the consensus protocol, and to address the limitations imposed by storage services out of our control.

The star topology imposed significant limitations on the system by preventing the nodes from communicating and collaborating directly, as is typical of distributed systems. All interactions had to pass through the client. In this structure, the client's role has shaped the logic of the consensus mechanism: being the only node with full connectivity to all replicas and the only component responsible for issuing update requests naturally determines its function within the consensus protocol. Despite acting as the sole coordinator, this work demonstrates that the principles of distribution and consensus can be applied in such a context. Rather than making autonomous decisions, the central node operates through a consensus algorithm that evaluates and reconciles information obtained from the replicas. The broader significance of this work lies in demonstrating how the properties of distributed systems can be leveraged in architectures that retain centralized features. These solutions are particularly relevant in enterprise-oriented scenarios.

Another defining feature of the system is that it uses external storage services rather than operating its own storage infrastructure. While this reduces dependence on a single infrastructure, it also limits the client's ability to verify that the servers are functioning correctly. Although the consensus protocol can tolerate crashes, it cannot address all unexpected behaviours that may arise in this setting. The reputation system mitigates this issue by enabling the monitoring of these independent servers' behaviour over time. Importantly, it also allows clients to choose which servers to rely on based on their reputation values. This reduces dependence on fixed server choices and reinforces the distributed nature of the system. Furthermore, since reputation information must be exchanged among multiple clients, the mechanism introduces peer-to-peer characteristics to the architecture.

This work helped to solve a specific problem posed by the host company. The research activity also provided insights into the behaviour of distributed solutions in centralized architectures, as well as the limitations that arise when applying consensus algorithms in such environments.

Appendix

This appendix reports the Python source code of the `consensus.py` module, which implements the core logic of the Consensus Algorithm presented in Chapter 4. This module contains the implementation of the update and restore phases, the handling of timeouts and retries, the management of unresponsive or unavailable servers, and the decision-making logic based on majority and weighted fallback.

The complete implementation of the protocol is available at `https://github.com/ilariapalumbo/distributed-consensus-star-topology.git`.

Consensus module

```python
import time
import hashlib

class ConsensusAlgorithm:
    def __init__(self, servers):
        """
  Represents the consensus algorithm logic
  params:
        servers: a list of Server objects forming the cluster
        """
        self.servers = servers
        self.unresponsive_servers = set()
        self.unavailable_servers = set()

    def validate_file(self, file):
        """
        Validates a file before it is sent to the servers
  Return True if the file is valid, otherwise False
  params:
        file: a File object
        """
        is_valid, message = file.is_valid()
        if not is_valid:
            print(f"Consensus: Validation failed - {message} \n")
            return False
        return True

    def update_consensus(self, file, timeout_ms=2, retry_limit=3, retry_period_ms=5):
        """
        Handles the update phase with a client-side timeout for ACKs
        Retries sending updates if ACKs are not received, respecting retry limits and periods
        Simulates server failures and recovery
  Return True if consensus was reached (all servers responded with ACKs), False otherwise
  params:
        file: the file to be updated
        timeout_ms: maximum time in milliseconds to wait for ACKs
        retry_limit: Maximum number of retries for unresponsive servers
        retry_period_ms: Period in milliseconds between retries for unresponsive servers
        current_time_ms: The current simulation time in milliseconds
        """
        print(f"Consensus: Starting update for file {file.file_name}, version {file.version}\n")
        current_time_ms = int(time.time() * 1000)
        remaining_servers = set(self.servers)  # Start with all servers
        retries = {server: 0 for server in remaining_servers}

        while remaining_servers:
          for server in list(remaining_servers):
              try:
                  if retries[server] >= retry_limit:
                      print(f"Consensus: Server {server.id} has reached max retries. Marking as unresponsive.\n")
                      self.unresponsive_servers.add(server)
                      remaining_servers.remove(server)
                      continue

                  # Send the update to the server
                  update_applied = server.update_file(file, current_time_ms)
                  if not update_applied:
                      print(f"Consensus: Server {server.id} failed to apply the update.\n")
                      retries[server] += 1
                      continue
                  start_time = time.time()

                  # Wait for ACK
                  while time.time() - start_time < timeout_ms / 1000.0:
                      ack = server.send_ack()
                      if ack:
                          print(f"Consensus: ACK received from Server {server.id}.\n")
                          remaining_servers.remove(server)
                          break
                      time.sleep(0.001)
                  else:
                      print(f"Consensus: Timeout waiting for ACK from Server {server.id}. Retrying...\n")
                      retries[server] += 1
              except Exception as e:
                  print(f"Consensus: Server {server.id} encountered an error: {e}\n")
                  retries[server] += 1

          if remaining_servers:
```

```python
                print(f"Consensus: Waiting {retry_period_ms} ms before next retry for remaining servers.\n")
                time.sleep(retry_period_ms / 1000)  # Convert ms to seconds for sleep

            # Log the current state of retries
            print(f"Consensus: Current retries: {[f'Server {s.id}: {retries[s]} retries' for s in remaining_servers]}\n")

            # Break if retry limits are exhausted
            if all(retries[s] >= retry_limit for s in remaining_servers):
                print("Consensus: Retry limit reached for all remaining servers.\n")
                break

        # Return True if all servers have sent ACKs (remaining_servers is empty)
        if not remaining_servers:
            print("Consensus: All servers responded with ACKs. Update successful.\n")
            return True

        # Return False if some servers are still unresponsive
        print("Consensus: Some servers did not respond with ACKs.\n")
        return False


    def retry_unresponsive_servers(self, file, long_retry_limit=5, retry_interval=0.02):
        """
        Periodically retries to update temporarily unavailable servers
  params:
        file: The file to be updated
        long_retry_limit: Maximum number of retries for unresponsive servers
        retry_interval: Time in seconds between retries
        """
```

```
      retries = 0
      while self.unresponsive_servers and retries <
 long_retry_limit:
          print(f"Consensus: Retry {retries + 1} for
 unresponsive servers: {[s.id for s in self.unresponsive_servers
 ]}\n")
          for server in self.unresponsive_servers.copy():
              try:
                  update_applied = server.update_file(file)
                  if update_applied:
                      ack = server.send_ack()
                      if ack:
                          self.unresponsive_servers.remove(
 server)
                          print(f"Consensus: Server {server.id}
 successfully updated.\n")
              except Exception as e:
                  print(f"Consensus: Server {server.id} is still
  unavailable: {e}\n")

          retries += 1
          if self.unresponsive_servers:
              print(f"Consensus: Waiting {retry_interval *
 1000:.2f} ms before next retry...\n")
              time.sleep(retry_interval)

      if self.unresponsive_servers:
          print(f"Consensus: Following servers are permanently
 unavailable: {[s.id for s in self.unresponsive_servers]}\n")
          self.unavailable_servers.update(self.
 unresponsive_servers)
          self.unresponsive_servers.clear()

  def restore_consensus(self, retry_limit=2, retry_period_ms=5):
      """
      Handles the restore phase with majority rule and weighted
 fallback
Return the file with the highest weight or consensus result
params:
      retry_limit: Maximum number of retries if consensus is not reached
      retry_period_ms: Time in milliseconds between retries
      """
      print("Consensus: Starting restore phase with weighted fallback.\n")

      remaining_servers = set(self.servers)
      retries = {server: 0 for server in remaining_servers}
      all_responses = []

      for attempt in range(retry_limit):
          print(f"Consensus: Attempt {attempt + 1} to retrieve files from servers.\n")
          responses = []

          for server in list(remaining_servers):
              response = server.retrieve_file()
              if response:
                  responses.append((server.weight, response)) # Include server weight
                  remaining_servers.remove(server)
              else:
                  retries[server] += 1
                  if retries[server] >= retry_limit:
                      print(f"Consensus: Server {server.id} marked as unavailable.\n")
                      self.unresponsive_servers.add(server)
                      remaining_servers.remove(server)

          all_responses.extend(responses)

          if not remaining_servers:
              break

          print(f"Consensus: Waiting {retry_period_ms} ms before retrying.\n")
          time.sleep(retry_period_ms / 1000.0)

      if not all_responses:
```

```python
            print("Consensus: No files retrieved from any server.\n")
            return None

        # Group files by hash and calculate weights
        weighted_files = {}
        for weight, response in all_responses:
            file_hash = hashlib.sha256(response["content"].encode()).hexdigest()
            if file_hash not in weighted_files:
                weighted_files[file_hash] = {"total_weight": 0, "file": response}
            weighted_files[file_hash]["total_weight"] += weight

        # Apply majority rule based on hash
        majority_file = None
        total_servers = len(self.servers) - len(self.unresponsive_servers)
        for file_hash, data in weighted_files.items():
            count = sum(1 for _, response in all_responses if hashlib.sha256(response["content"].encode()).hexdigest() == file_hash)
            if count > total_servers / 2:  # Majority rule
                majority_file = data["file"]
                print(f"Consensus: Majority file selected: {majority_file['file_name']} (Hash: {file_hash}).\n")
                return {
                    "version": majority_file["version"],
                    "content": majority_file["content"],
                    "file_name": majority_file["file_name"],
                }

        # Fallback to highest weight if no majority
        most_weighted_file = max(weighted_files.values(), key=lambda x: x["total_weight"])
        print(f"Consensus: No majority. Falling back to file with highest weight: {most_weighted_file['file']['file_name']}, Total weight: {most_weighted_file['total_weight']}.\n")
        return {
            "version": most_weighted_file["file"]["version"],
            "content": most_weighted_file["file"]["content"],
            "file_name": most_weighted_file["file"]["file_name"],
        }
```

Listing A.1: Consensus mechanism

Bibliography

[1] Miguel Castro and Barbara Liskov. «Practical Byzantine Fault Tolerance». In: *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI).* USENIX Association, 1999, pp. 173–186.

[2] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. «Consensus in the Presence of Partial Synchrony». In: *Journal of the ACM* 35.2 (1988), pp. 288–323. DOI: `10.1145/42282.42283`.

[3] Michael J. Fischer. «The Consensus Problem in Unreliable Distributed Systems». In: *Proceedings of the International Conference on Foundations of Computation Theory (FCT).* Ed. by M. Karpinski. Vol. 158. Lecture Notes in Computer Science. Borgholm, Sweden: Springer, 1983, pp. 127–140. DOI: `10.1007/BFb0026484`.

[4] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. «Impossibility of Distributed Consensus with One Faulty Process». In: *Journal of the ACM* 32.2 (1985), pp. 374–382. DOI: `10.1145/3149.214121`.

[5] Fangyu Gai et al. «Proof of Reputation: A Reputation-Based Consensus Protocol for Peer-to-Peer Network». In: *Database Systems for Advanced Applications (DASFAA 2018).* Vol. 10828. Lecture Notes in Computer Science. ISBN 978-3-319-91457-2. Cham: Springer International Publishing, 2018, pp. 666–681. DOI: `10.1007/978-3-319-91458-9_41`. URL: `https://doi.org/10.1007/978-3-319-91458-9_41`.

[6] David K. Gifford. «Weighted Voting for Replicated Data». In: *Proceedings of the Seventh Symposium on Operating Systems Principles (SOSP)* (1979). Stanford University and Xerox Palo Alto Research Center, pp. 150–162.

DOI: 10.1145/800215.806583. URL: https://doi.org/10.1145/800215.806583.

[7] Seth Gilbert and Nancy Lynch. «Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services». In: *ACM SIGACT News* 33.2 (2002), pp. 51–59. DOI: 10.1145/564585.564601.

[8] Stan Gurtler and Ian Goldberg. «SoK: Privacy-Preserving Reputation Systems». In: *Proceedings on Privacy Enhancing Technologies* 2021.1 (2021), pp. 107–127. DOI: 10.2478/popets-2021-0007.

[9] Audun Jøsang and Roslan Ismail. «The Beta Reputation System». In: *15th Bled Electronic Commerce Conference: e-Reality: Constructing the e-Economy.* Bled, Slovenia, June 2002.

[10] Audun Jøsang, Roslan Ismail, and Colin A. Boyd. «A survey of trust and reputation systems for online service provision». In: *Decision Support Systems* 43.2 (2007), pp. 618–644. DOI: 10.1016/j.dss.2005.05.019.

[11] Leslie Lamport. «How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs». In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691. DOI: 10.1109/TC.1979.1675439.

[12] Leslie Lamport. «Paxos Made Simple». In: *ACM SIGACT News* 32.4 (2001), pp. 18–25. DOI: 10.1145/568425.568433.

[13] Leslie Lamport. «Time, Clocks, and the Ordering of Events in a Distributed System». In: *Communications of the ACM* 21.7 (1978), pp. 558–565. DOI: 10.1145/359545.359563.

[14] Leslie Lamport, Robert Shostak, and Marshall Pease. «The Byzantine Generals Problem». In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (July 1982), pp. 382–401. DOI: 10.1145/357172.357176. URL: https://doi.org/10.1145/357172.357176.

[15] Diego Ongaro and John Ousterhout. «In Search of an Understandable Consensus Algorithm (Extended Version)». In: *Proceedings of the USENIX Annual Technical Conference.* USENIX Association, 2014, pp. 305–319.

[16] Fred B. Schneider. «Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial». In: *ACM Computing Surveys* 22.4 (1990), pp. 299–319. DOI: `10.1145/98163.98167`.

[17] Gayatri Swamynathan, Kevin C. Almeroth, and Ben Y. Zhao. «The Design of a Reliable Reputation System». In: *Electronic Commerce Research* 10.3–4 (2010). Open access, pp. 239–270. DOI: `10.1007/s10660-010-9064-y`. URL: `https://doi.org/10.1007/s10660-010-9064-y`.

[18] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. 2nd. Prentice Hall, 2007. ISBN: 978-0-13-239227-3.

[19] Bin Yu and Munindar P. Singh. «Distributed Reputation Management for Electronic Commerce». In: *Computational Intelligence* 18.4 (2002), pp. 535–549.

[20] Jun Zou et al. «A Proof-of-Trust Consensus Protocol for Enhancing Accountability in Crowdsourcing Services». In: *IEEE Transactions on Services Computing* (2018). To appear in a future issue; accepted for publication. DOI: `10.1109/TSC.2018.2823705`. URL: `https://doi.org/10.1109/TSC.2018.2823705`.