Distributed System Project Report

Nicolò Fadigà
258786
University of Trento
nicolo.fadiga@studenti.unitn.it

Alberto Messa
258855
University of Trento
alberto.messa@studenti.unitn.it

Abstract

This report presents the **Mars Distributed Storage System**, a peer-to-peer key-value network developed as a proof-of-concept for the Distributed Systems course project. The system simulates a network of data collection stations across Mars, gathering scientific data from rovers, satellites, and environmental sensors.

Designed for resilience and fault-tolerance, the system ensures data availability despite node failures or network disruptions. Each node acts as an autonomous actor, storing data, coordinating replication, and handling dynamic membership events, while maintaining sequential consistency.

The report details the architectural choices, system operations, replication mechanisms, and management of dynamic events such as node joins, leaves, crashes, and recoveries, demonstrating the system's ability to provide a robust and consistent distributed storage service in an unreliable environment.

1. Introduction

The **Mars Distributed Storage System** project aims to develop a fault-tolerant distributed key-value store suitable for extreme and unpredictable environments, such as the Martian surface. The system is designed to handle multiple concurrent clients, node crashes, dynamic membership changes, and network delays.

The network is structured as a ring of nodes, each acting as an Akka actor responsible for local storage, message routing, and replication. Every node knows the current set of participants in the ring and can directly serve requests for any key. Clients perform `read` and `write` operations through a node that acts as a coordinator, managing replication and quorum logic. The system automatically redistributes data when nodes join or leave, ensuring that all data remains assigned according to the ring structure. It also handles node crashes and recoveries, maintaining sequential consistency through versioning.

System Assumptions. The system relies on a set of assumptions that simplify its design while highlighting its core functionality:

1. The network is reliable and FIFO.
2. Membership or failure events occur one at a time, and no other operations are ongoing during these events.
3. Failures do not occur during the processing of requests within the timeout window.
4. Crash detection is not required.
5. Clients execute commands one at a time, waiting for the coordinator's reply before sending the next request.
6. Keys are unsigned integers, and data items are strings without spaces.
7. Replication parameters (`N`, `R`, `W`, and `T`) are configurable.

Each section of this report elaborates on specific aspects of the system: architecture, replication and quorums, node lifecycle management, sequential consistency, data partitioning, implementation details, and the assumptions that guided the design. This structure highlights the rationale behind our design choices and explains the mechanisms that ensure the system functions correctly under challenging conditions.

2. Project Structure

The implementation of the Mars Distributed Storage System follows a modular actor-based architecture built on Akka. The system is organized into four main components that together define the control flow, data management, and client interaction mechanisms: `Main`, `TestManager`, `Client`, and `Node`.

The `Main` class serves as the entry point of the application. It is responsible for loading configuration parameters, initializing the actor system, and starting the distributed environment. It also triggers a sequence of test scenarios that evaluate the system's behavior under different conditions, such as concurrent access, node crashes, recoveries, and membership changes.

The `TestManager` serves as the central coordination actor and a key architectural component. Inspired by Laboratory 3 (Akka Virtual Synchrony), where a centralized actor was permitted to simplify view changes, this design reproduces the same concept: the `TestManager` maintains the list of active and crashed nodes, propagates membership updates, and ensures that all actors operate under a consistent system view.

In addition to coordination, the `TestManager` enforces the assumption stated in Item 2: *"Membership or failure events occur one at a time, and no other operations are ongoing during these events."* To satisfy this constraint, it must track all ongoing operations, including both membership-related actions and client requests. Therefore, the `main` program forwards all planned actions to the `TestManager`, which then dispatches them to the appropriate actors. Internally, boolean flags indicate whether a client request is in progress or a view change is underway, preventing overlap between these operations. This mechanism ensures safe transitions, preserves a stable configuration, and maintains sequential consistency.

Each `Client` actor represents an external user of the distributed storage system. Clients issue read and write requests to nodes, one operation at a time, and wait for responses before sending new commands. The assumption that clients behave sequentially and independently simplify coordination while preserving realistic interaction patterns.

Finally, each `Node` embodies a peer in the distributed key-value network. Nodes handle local data storage, replication, and coordination logic according to quorum parameters. Each node can serve as a coordinator for client requests and as a replica for other nodes, ensuring fault tolerance and consistency. The ring-based topology allows each node to be aware of all other participants and to autonomously handle events such as joins, leaves, and recoveries, triggering automatic data redistribution when needed.

3. Design Choices

3.1 Tracking of Ongoing Operations

To correctly coordinate client operations with membership events, the system employs a centralized tracking mechanism implemented within the `TestManager`. Since this actor orchestrates all join, leave, crash, and recovery actions, it must always be aware of whether any client request is currently active. For this purpose, the `TestManager` maintains the `activeClientRequests` structure, a mapping from client identifiers to their pending operations (e.g., `‘‘READ key’’`, `‘‘WRITE key value’’`).

In addition, a boolean flag `isViewChangedStable` records whether the system is undergoing a membership transition. This mechanism guarantees compliance with the system assumption that no membership or failure event may overlap with an ongoing client request.

Each node also keeps local information regarding operations for which it acts as coordinator. Write operations are tracked through the per-key structure `writeRequestList`, whereas read operations are monitored in `readRequestList`, indexed by the pair `(key, clientName)`. This design enables nodes to handle concurrent requests safely and maintain consistency even during coordinator changes or message delays.

3.2 Main Messages Used in the System

The system relies on a structured set of message types exchanged among the `TestManager`, the `Client` actors, and the network `Node` actors. These messages coordinate client operations, node actions, membership changes, and data replication during the execution of the protocol.

Messages handled by the TestManager. The `TestManager` primarily handles coordination messages originating from the main program: `ClientRequest` for read/write operations, which it forwards to the clients and monitors via the corresponding `ClientResponse`; `NodeActionRequest`, sent directly to nodes to trigger membership events like join, leave, crash, or recover, and tracked through the respective `NodeActionResponse`.

At the completion of each operation, the `TestManager` decrements a latch used by the main program; once the latch reaches zero, the main process knows that the current test scenario has fully terminated.

Messages handled by the Client. The `Client` actor primarily processes messages related to read and write operations: `Write` and `Read` requests originating from the `TestManager`, and their corresponding `WriteResponse` and `ReadResponse` messages returned upon operation completion. To ensure the client always addresses active participants, the `TestManager` sends `UpdateNodeList` messages whenever the ring membership changes, allowing the client to maintain an accurate view of the available network nodes.
[1]

Messages handled by the Nodes. Each `Node` processes messages according to its current state: active, joining/recovering, or crashed. In the active state, the node handles `ReadRequest` and `WriteRequest` messages from clients, performs version queries (`GetVersionRead`, `GetVersionWrite`) and applies updates with `UpdateValue` to maintain the local key-value store. Membership-related actions are triggered via `NodeAction` messages from the `TestManager`, while network bootstrap and data transfer are managed through `GetPeers`, `PeerResponse`, `ItemRequest`, `DataItemsBatch`, `AnnounceJoin`, `AnnounceLeave`, and `TransferData` messages.

During joining or recovery, the node additionally handles messages such as `JoinTimeout`, `PeerResponse`, and `DataItemsBatch` to synchronize with the current network state. In the crashed state, nearly all messages are ignored except for `RecoverTimeout` and `NodeAction`, which trigger the recovery procedure.

3.3 Crash and Recovery Simulation

Crash and recovery are simulated explicitly through messages sent from the `TestManager` to the target node. A crash is modelled as a local transition of the node into a restricted state in which all messages are ignored except for the recovery request and its associated timeout. Unlike real distributed systems, no crash detection mechanism is implemented, in accordance with the project assumptions: crashed nodes remain part of the network view and may still be contacted by clients, but they simply do not reply.

Recovery is more elaborate, as it must reintroduce the node into a consistent system state. When a recovery action is triggered, the `TestManager` provides the recovering node with a bootstrap reference, selected from the current network (either active or crashed). The node then attempts to contact the bootstrap to obtain the latest membership view; if this step fails within the timeout window, the recovery procedure is aborted.

Once the node acquires the current network configuration, it determines its correct position in the ring and requests state information from its predecessor and successor. These nodes necessarily hold all key–value pairs for which the recovering node is responsible, even if membership has changed during its downtime. The recovering node receives these batches of keys and performs a maintenance read for each of them to reconstruct its local store consistently. When this synchronization is complete, the node transitions back to the normal active state and resumes processing client operations.

4. Reasoning on implementations

4.1 Sequential Consistency Guarantees

Our system guarantees *sequential consistency* through a combination of quorum rules and per-key locking mechanisms. The basic quorum requirement ensures that any read or write operation involves a sufficient number of replicas to respect the parameters R and W, according to the constraints:

$$R + W > N \quad \text{and} \quad W > N/2,$$

[1]The client does not participate in coordination or consistency mechanisms. It maintains a local view of currently available nodes solely to select a reachable node to contact for read and write requests. This information is provided by the TestManager and does not affect replication, quorum logic, or correctness, which are entirely handled by storage nodes acting as coordinators.

so that at least one replica with the **latest version** participates in every quorum. This alone, however, **does not fully guarantee sequential consistency** in the presence of *overlapping operations*.

Consider a scenario where a write request is in progress and its coordinator has reached the quorum with the new version (say version 2), while other replicas still hold version 1. If a read request overlaps with this write and reaches a quorum of updated replicas, it return the **new value**. However, a subsequent read on the same key might reach a quorum of non-updated replicas and return the **older version**, effectively causing the system to jump back in time and **breaking sequential consistency** (see Figure 1). This illustrates the core problem of concurrent **W**–**R** and **W**–**W** interactions.

To prevent this, our implementation introduces **per-key locks** at the replica level, used only for write operations. Each node maintains a **write vote** for each key, which can be granted to a single coordinator at a time. When a write is in progress, the node locks the key for that coordinator, effectively serializing all operations on that key. During the lock period, the node will **reject any other write or read** request for that key, ensuring that no client can observe an intermediate or partially applied update.

Once the write completes or times out, the lock is released, allowing subsequent operations. This mechanism may reduce concurrency-operations on a key are serialized while a write is active-but it **guarantees sequential consistency**: every read sees the latest committed value, and old versions are never observed once a new value has been written.

4.2 Message Efficiency in Read/Write and Network Operations

Our strategies for read and write operations use the minimum number of messages necessary to guarantee correctness. Excluding the communication between the client and the coordinator node, reads involve only version requests to replicas and their responses, which suffice to determine the latest value. Writes additionally conclude with either a success message confirming the update or a failure message after a timeout.

For join operations, the protocol is slightly less minimal but remains efficient. The joining node first contacts a bootstrap node to obtain the current network membership, then requests from its immediate successor the set of keys for which it will be responsible. The successor responds with the relevant key identifiers, and the joining node performs a maintenance phase (essentially a sequence of read operations on those keys, i.e. a version request followed by the corresponding response), before announcing itself as fully active.

Leave operations are already performed with minimal messaging: the departing node computes the new active network, transfers its keys to the appropriate successor nodes, and informs the remaining nodes of its departure so that they can update their local state. This approach avoids any redundant communication while ensuring that data remains correctly distributed and that all nodes maintain an up-to-date view of the network.

4.3 Item Repartitioning Logic

To correctly manage data distribution during these operations, the system employs a deterministic algorithm based on the ring topology. Given a key and the current list of active nodes, the system identifies the N responsible replicas by selecting the closest nodes in clockwise order. This logic ensures that repartitioning is handled consistently across all scenarios:

- **Join:** The joining node uses this logic to identify exactly which keys it must acquire from its successor. Simultaneously, existing nodes calculate which keys they are no longer responsible for and prune them to free up storage.
- **Leave:** Before disconnecting, the leaving node calculates the new set of responsible replicas for its data. It transfers items strictly to those specific nodes, ensuring data is not sent to irrelevant peers.
- **Recovery:** Upon recovery, a node filters its local storage using the current view of the network, discarding any items that were reassigned to other nodes while it was offline.

By strictly following this mapping logic, the system guarantees that data is always stored by the correct nodes and that transfers occur only when strictly necessary.

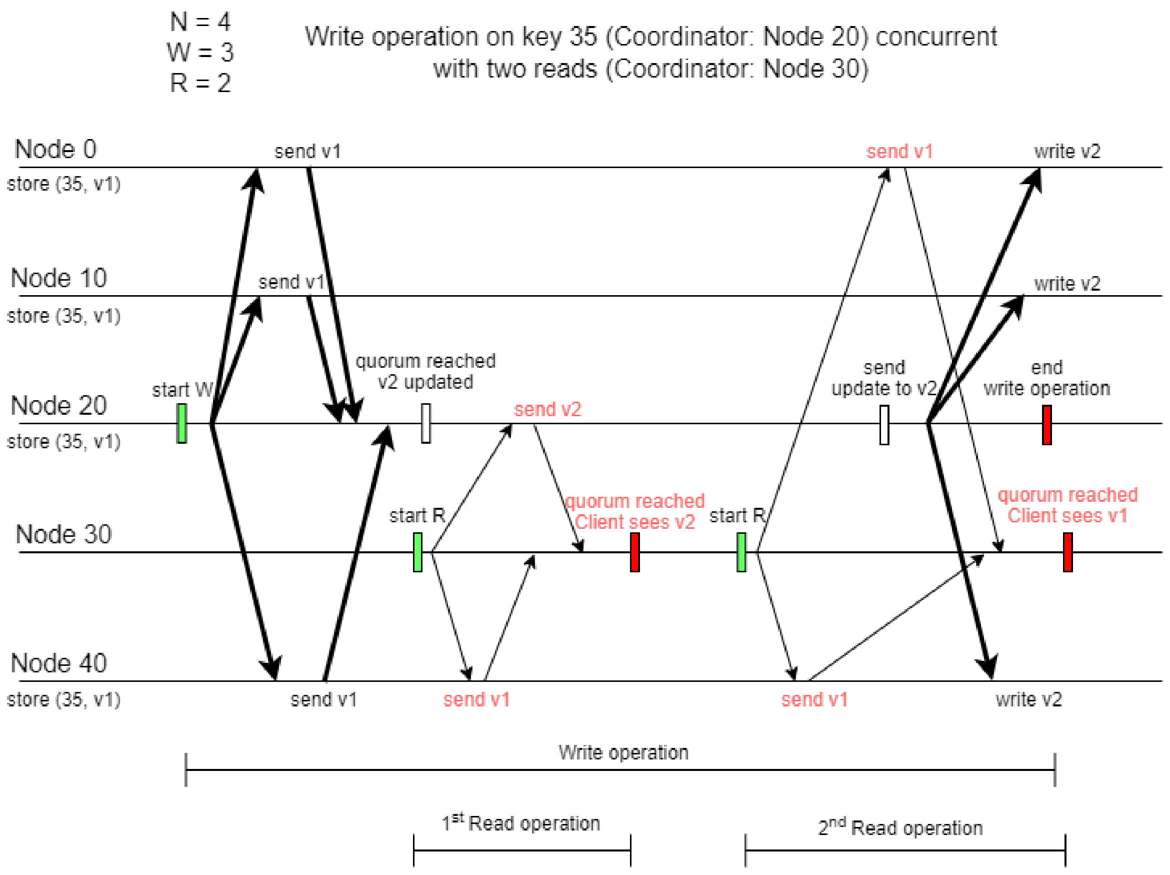

Figure 1: Scenario demonstrating a violation of sequential consistency. A read operation observes the new version (v2), but a subsequent overlapping read returns an older version (v1), effectively causing the system to jump back in time.