Distributed Voting and Polling System

Final Report for the Distributed Systems Course 2025/2026

Roberto Pisu
roberto.pisu@studio.unibo.it

April 15, 2026

Contents

Abstract

This project presents a distributed voting and polling system designed to operate in a decentralized and fault-tolerant environment. The system allows users to create polls and cast votes through a REST-based web service, while maintaining replicated application state across multiple nodes.

The core challenge addressed is maintaining replicated poll state in the presence of concurrent updates, node crashes, and network partitions. To address this, the system adopts a Conflict-free Replicated Data Type (CRDT) model, allowing nodes to process updates independently while guaranteeing eventual convergence of replicas [4]. Update dissemination is implemented through a best-effort replication strategy combined with periodic anti-entropy synchronization, ensuring that updates missed during temporary disconnections are eventually propagated.

To improve dependability, each node persists updates using a write-ahead log (Write-Ahead Log (WAL)) and periodically creates checkpoints, allowing recovery after crashes with no loss of locally persisted updates. A heartbeat-based failure suspicion mechanism is also included to monitor node reachability and identify potentially failed or disconnected peers.

The system prioritizes availability and partition tolerance over strong consistency, making it suitable for distributed environments where network reliability cannot be guaranteed [3, 5]. The final implementation demonstrates how fundamental distributed systems concepts such as replication, consistency trade-offs, and fault tolerance can be applied in a practical application.

1 Concept

The project implements a distributed polling and vote aggregation system exposed as a RESTful web service, optionally accessible through a simple web-based user interface.

Each node in the system exposes the same Application Programming Interface (API) and maintains a local replica of the application state.

1.1 Type of product

The system can be classified as:

- a **web service**, exposing HTTP endpoints for interaction
- a **distributed application**, composed of multiple cooperating nodes

1.2 Use case

The system allows users to create polls, cast votes on predefined options, and retrieve current results. Each vote is processed by a node and later propagated to the other nodes in the system.

- **What**: users can create polls through implicit instantiation on the first vote, submit additional votes, and query results.
- **Where**: users may be geographically distributed and interact with any reachable node
- **When**: interactions are sporadic but may occur concurrently from multiple users
- **How**: users interact via HTTP requests (e.g., REST API or web interface)
- **Data**: the system stores poll identifiers, poll options, and vote counters
- **Roles**: the system assumes a single user role, without authentication or authorization mechanisms

1.3 Why distribution is needed

The system is distributed in order to address several relevant requirements:

- **Fault tolerance**: the system continues to operate even if some nodes fail or become temporarily unreachable
- **Availability**: users can interact with any reachable node, even during partial failures or network partitions
- **Scalability**: multiple nodes can share the workload as the number of requests increases
- **Geographical distribution**: nodes can be deployed in different locations to reduce latency for users

2 Requirements Elicitation and Analysis

This section identifies the requirements of the distributed voting and polling system, focusing on **what** the system must provide, independently of implementation details.

2.1 Functional Requirements

- **FR1 – Poll creation**
  The system shall allow a poll to become available in the distributed state under a unique poll identifier. In the current project, poll creation is not exposed through a dedicated endpoint; instead, a poll and its options are implicitly initialized when the first valid vote referencing them is submitted.
  Acceptance criterion: after the first valid vote for a previously unseen poll identifier and option, the poll becomes retrievable from the system and appears in the poll listing.

- **FR2 – Vote submission**
  The system shall allow users to submit a vote increment for a specific option within a poll. In the current prototype, vote submission updates aggregated counters and does not enforce voter identity, uniqueness, or revocation constraints.
  Acceptance criterion: a user sends a vote request and receives confirmation that the vote has been registered locally.

- **FR3 – Poll result retrieval**
  The system shall allow users to retrieve the current vote counts for a given poll.
  Acceptance criterion: a request to a poll endpoint returns the current aggregated counts for each option.

- **FR4 – Poll listing**
  The system shall provide a list of available poll identifiers.
  Acceptance criterion: a request to the poll listing endpoint returns all existing poll IDs.

- **FR5 – Distributed operation**
  The system shall allow users to interact with any node in the system.
  Acceptance criterion: poll creation, vote submission, and query operations succeed when executed on any reachable node.

- **FR6 – Eventual propagation of updates**
  The system shall propagate poll and vote updates across nodes.
  Acceptance criterion: after sufficient time and in the absence of permanent communication failures, all nodes receive the same updates.

2.2 Non-Functional Requirements

- **NFR1 – Availability**
  The system shall remain operational even if some nodes are unreachable.

Acceptance criterion: users can still submit votes and query results through reachable nodes during partial failures.

- **NFR2 – Partition tolerance**
  The system shall tolerate network partitions between nodes.
  Acceptance criterion: nodes continue operating independently when disconnected and later reconcile their state after communication is restored.

- **NFR3 – Eventual consistency**
  The system shall guarantee that, in the absence of permanent communication failures and after sufficient synchronization time, all replicas eventually converge to equivalent poll states.
  Acceptance criterion: after communication is restored and anti-entropy has had enough time to run, all nodes expose the same aggregated counts for the same polls.

- **NFR4 – Durability**
  The system shall persist local updates to stable storage so that acknowledged local changes are not lost after a crash.
  Acceptance criterion: after a node restart, all previously persisted local updates are still reflected in the reconstructed state.

- **NFR5 – Fault recovery**
  The system shall recover node state after a crash and resume normal participation in the distributed system
  Acceptance criterion: once restarted, a node reloads its state from persistent storage and continues exchanging updates with peers without manual reinitialization.

- **NFR6 – Concurrency support**
  The system shall correctly handle concurrent updates performed on different nodes.
  Acceptance criterion: concurrent vote submissions are merged without conflicts or lost updates.

- **NFR7 – Performance (best-effort)**
  The system shall respond to user requests within a reasonable time under normal operating conditions.
  Acceptance criterion: poll creation, vote submission, and query operations complete with short response times in the absence of severe overload or prolonged network failures.

2.3 Implementation Constraints

- **IC1 – Programming language**
  The system shall be implemented using Python.
  Justification: consistency with course tools and suitability for rapid prototyping.

- **IC2 – Communication protocol**
  The system shall use HTTP-based communication for client-node and inter-node interaction.
  Justification: simplicity, readability, and interoperability.

- **IC3 – Deployment environment**
  The system shall be deployable using containerization tools (e.g., Docker).
  Justification: ease of deployment, reproducibility, and support for distributed testing scenarios.

2.4 Relevant Distributed System Features

This section highlights the distributed systems features that are most relevant to the project.

- **Transparency**
  The system provides partial transparency. Users interact with any reachable node through a uniform REST interface without needing to be aware of inter-node communication or replication. Full transparency is not achieved, however, because temporary inconsistencies may be visible under the eventual consistency model.

- **Replication, partition tolerance, and eventual consistency**
  The application state is replicated across multiple nodes, each of which can process client requests independently. This supports operation under network partitions and avoids dependence on a single central node. Since updates are propagated asynchronously, replicas may temporarily diverge, but they are designed to converge again after communication is restored and synchronization progresses.

- **Fault tolerance and recovery**
  Fault tolerance is a primary concern of the project. The system is designed to remain operational even if some nodes fail or become unreachable. Data durability and recovery are supported through write-ahead logging and periodic checkpointing, allowing a crashed node to reconstruct its local state after restart and then resume synchronization with the rest of the cluster [2].

- **Resource sharing**
  Nodes share replicated application state representing polls and vote counts. Coordination is achieved through distributed synchronization rather than centralized control.

- **Openness and interoperability**
  The system relies on standard web technologies, namely HTTP and JavaScript Object Notation (JSON), which improves openness and facilitates interoperability among heterogeneous components.

- **Limited security scope**
  Security is intentionally simplified in this prototype. The system does not implement full user-level authentication or fine-grained authorization, and assumes a mostly trusted environment. Nevertheless, basic input validation is performed to preserve data integrity and reject malformed requests.

- **Evolvability and maintainability**
  The modular organization of the system supports future extensions, such as stronger security mechanisms, richer poll models, or alternative replication strategies.

3 Design

This chapter describes the design choices adopted to satisfy the requirements identified in section 2 and the distributed-system features discussed in section 2.4. The final prototype is consistent with the design choices discussed in this chapter: a cluster of homogeneous nodes exposing the same REST API, each maintaining a local replica of the application state and cooperating through asynchronous replication and periodic synchronization.

The design is driven by the need to preserve **availability**, **partition tolerance**, **fault recovery**, and **eventual consistency**, while keeping the architecture simple enough for a lightweight implementation and reproducible deployment.

3.1 Architecture

The system adopts a decentralized architecture composed of multiple homogeneous nodes, each maintaining a full replica of the application state.

Clients can interact with any reachable node, which processes requests locally without relying on a central coordinator. Each node independently updates its local state and propagates changes to its peers through asynchronous communication.

No primary node or consensus layer is introduced. Instead, the system relies on state replication and eventual convergence mechanisms to maintain consistency across replicas.

This architectural choice directly reflects the requirements discussed in Sections 1 and 2, particularly with respect to availability, partition tolerance, and decentralized request handling.

3.2 Infrastructure

The infrastructure is intentionally lightweight but no longer consists only of the replicated application nodes. In the current version, the deployment includes the following components:

- **client(s)**: external users interacting through HTTP requests, either directly or through a browser-based interface;

- **reverse proxy**: an Nginx container exposing a single public entry point for manual interaction with the cluster;
- **distributed voting nodes**: multiple identical application instances, each running the same service logic;
- **persistent local storage**: one Docker volume per node, used to store checkpoint and WAL files.

Each voting node remains self-contained with respect to application logic: it handles client requests, stores its local replica, persists recovery information, and communicates directly with peer nodes for replication, heartbeat exchange, and anti-entropy. However, the cluster includes an additional infrastructural component in front of the nodes, namely a reverse proxy used to simplify access to the web interface and to avoid exposing many application endpoints in the default deployment mode.

The nodes are deployed as separate Docker containers. A helper script dynamically generates both:

- a Docker Compose configuration for a configurable number of nodes;
- an Nginx configuration file exposing one proxied path per node.

For each node, the generated deployment defines:

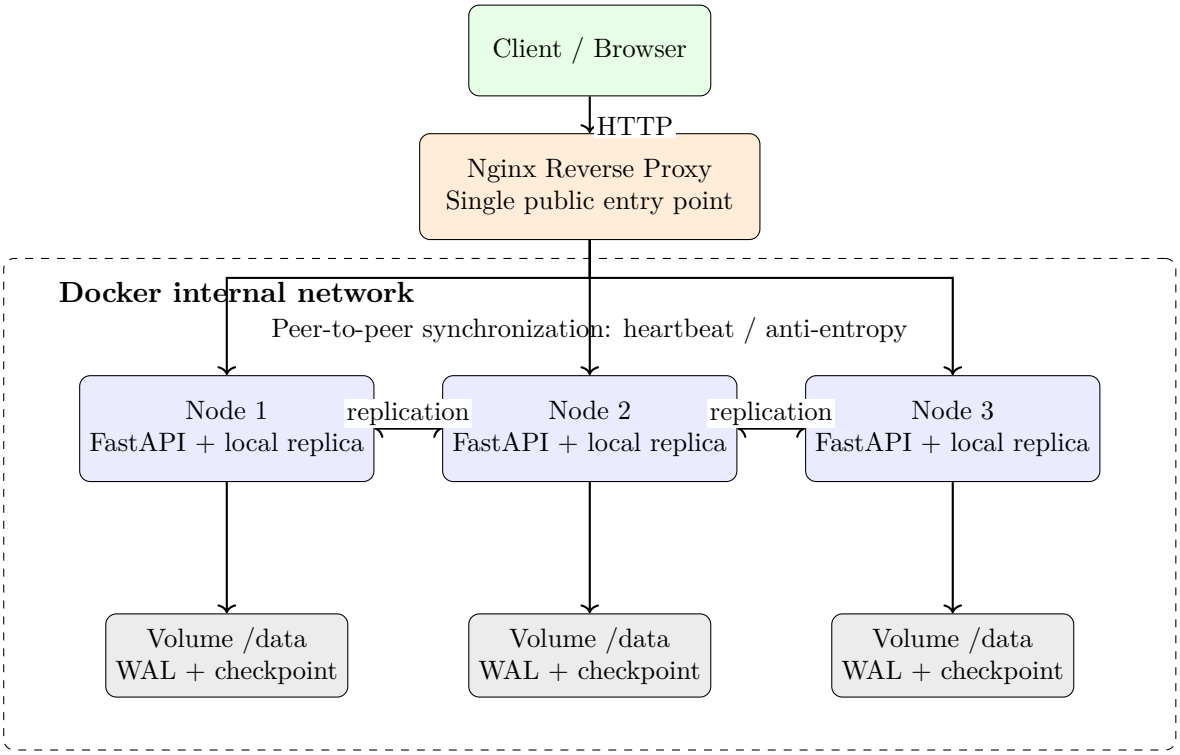
- a unique logical identity (e.g., `node1`, `node2`, `node3`);
- an internal HTTP port;
- a list of peer node addresses;
- a dedicated persistent volume mounted at `/data`.

For the proxy, the generated configuration exposes:

- a root page on the proxy itself;
- one path of the form `/node/i/` forwarding requests to node `i`.

The distribution model is therefore a containerized cluster on a shared Docker network. In the usual setup, all containers run on the same host for testing and demonstration purposes, but the architecture does not conceptually depend on physical co-location: the same node design could be deployed on distinct machines as long as they can reach each other over HTTP.

Node discovery is still static. Each node receives the list of peers through environment variables generated at deployment time, so service discovery is configuration-based rather than dynamic. The reverse proxy is also generated statically from the same source of truth, which keeps the public access paths aligned with the actual cluster topology.



3.3 Modelling

The main domain entities are the following:

- **Poll**: identified by a unique poll identifier;
- **Option**: one selectable choice within a poll;
- **Vote**: an increment of the counter associated with one option of one poll;
- **Node**: a replica that stores and updates part of the distributed counter state.

In the current implementation, a poll is not created through an explicit dedicated command. Instead, it is created **implicitly** when the first vote is submitted for a previously unseen `poll_id`. Likewise, an option is created implicitly when first referenced by a vote.

The core state of the system is modelled as a Conflict-free Replicated Data Type (CRDT)-based nested map [4]:

```
g_counter[poll_id][option][node_id] = value
```

This means that, for each poll and option, the system stores one counter component per node. The total number of votes for an option is obtained by summing all node-local components for that option.

The main domain events are:

- **vote submitted**: a client sends a vote to a node;
- **internal update received**: a node receives a replicated CRDT component update from a peer;
- **anti-entropy merge**: a node periodically compares its local state with a peer's full state and merges newer components;
- **checkpoint created**: a node persists a full snapshot of its local replica;
- **node restarted**: a node reconstructs its state from checkpoint and WAL.

The exchanged messages can be classified as:

- **client commands**: vote submission requests;
- **client queries**: poll listing, poll retrieval, status inspection;
- **internal update messages**: propagation of individual counter updates;
- **internal state transfer messages**: full poll state or full cluster state used during synchronization;
- **heartbeat messages**: liveness signals exchanged among peer nodes.

Semantically, the system does not store individual voter identities or ballots. Instead, each submitted vote is represented as a monotonic increment of a CRDT counter associated with a poll option. For this reason, the prototype is more precisely a distributed polling and vote aggregation system. This design is intentional, as it allows conflict-free replication and strong eventual consistency under concurrent updates and network partitions [4, 5].

3.4 Interaction

Two main categories of interaction are present in the system.

Client–node interaction. Clients communicate with any reachable node through HTTP using a REST-style interface. The main external endpoints are:

- `POST /vote`: submit a vote for a poll option;
- `GET /polls`: retrieve the list of known poll identifiers;
- `GET /poll/{poll_id}`: retrieve the aggregated results for a poll;
- `GET /status`: inspect the liveness state of peer nodes as seen by the current node.

Node–node interaction. Nodes communicate with one another through internal HTTP endpoints. These interactions are asynchronous and best-effort. The main internal patterns are:

- **asynchronous update propagation**: when a node processes a vote, it generates the corresponding CRDT update locally and then forwards it to peers;
- **periodic anti-entropy**: nodes periodically exchange state information and merge any missing updates;
- **heartbeat exchange**: nodes periodically ping each other to estimate peer liveness.

The most relevant interaction patterns enacted by the system are therefore:

- **request-response** for client operations and internal HTTP exchanges;
- **best-effort asynchronous dissemination** for update propagation;
- **periodic reconciliation** for eventual convergence;
- **heartbeat-based liveness monitoring** for failure suspicion.

A typical vote submission sequence is the following:

1. a client sends a vote request to one node;
2. the node processes the request locally;
3. the node returns a response to the client;
4. the node asynchronously propagates the corresponding update to its peers.

3.5 Behaviour

Each node behaves as a **stateful component**: it maintains a local CRDT replica, accepts client requests, incorporates updates received from peers, and periodically synchronizes with the rest of the cluster.

At startup, a node reconstructs its durable local state by loading the most recent checkpoint and replaying the updates recorded in the WAL. Once recovery is completed, it starts the background tasks responsible for heartbeat exchange, anti-entropy synchronization, and periodic checkpoint creation.

When a vote is received, the node computes the corresponding CRDT update for the selected poll and option, durably records it, applies it to the local replica, and then asynchronously propagates the update to its peers.

When a replicated update is received from another node, the receiver checks whether that update carries information not yet reflected in its local state. Stale or duplicate

updates are ignored; otherwise, the update is recorded and merged into the local replica according to the CRDT rule.

As a consequence, each node independently handles the requests it receives locally while progressively incorporating remote updates from the other replicas. State evolution is monotonic: counters only increase, and merge operations never remove information.

3.6 Data and Consistency Issues

The system stores both **application state** and **recovery metadata**.

Application state. The application state consists of the distributed voting counters, represented as a nested key-value structure:

- poll identifier;
- option identifier;
- node identifier;
- integer counter value.

This choice is suitable because the domain is simple, hierarchical, and naturally represented through maps rather than through relations or graphs. No external Database Management System (DBMS) is used: each node stores its state in memory and periodically serializes it to disk.

Persistent recovery data. Each node stores:

- a **checkpoint file**, containing a full serialized snapshot of the current replica state;
- a **WAL file**, containing one JSON record per update.

This persistence design was chosen because it supports durability and efficient crash recovery with limited implementation complexity.

The queries on persistent data are local to the node and happen mainly:

- at startup, to restore checkpoint and WAL;
- during checkpoint creation, to write the current state;
- during WAL truncation after a successful checkpoint.

Read and write concurrency is possible at the application level because client requests and background tasks may overlap. For this reason, the implementation introduces explicit locks to serialize durability-critical operations and protect in-memory state consistency.

The data shared between components is the replicated CRDT state. This sharing is necessary because all nodes must eventually converge toward equivalent poll results despite concurrent updates and temporary communication failures.

3.7 Design Trade-offs and Rejected Alternatives

The design of the system is based on a set of explicit trade-offs aimed at balancing consistency, availability, fault tolerance, and implementation complexity. Rather than pursuing strong consistency through coordination-heavy mechanisms, the system adopts a decentralized approach based on Conflict-free Replicated Data Types (CRDTs), favoring robustness and simplicity in the presence of failures and network partitions.

Consistency model: CRDT vs. strong consistency The system adopts a state-based CRDT model, specifically a G-Counter-style replicated structure, to guarantee eventual consistency without requiring synchronous coordination between nodes.

An alternative approach would have been to enforce strong consistency through distributed locks, leader-based replication, or consensus protocols such as Paxos or Raft. This alternative was rejected because it would significantly increase implementation complexity and reduce availability during network partitions. The chosen CRDT-based approach allows each node to process updates independently and guarantees convergence after synchronization, at the cost of temporary replica divergence.

Persistence strategy: local WAL and checkpoint vs. centralized storage Each node persists its own state locally using write-ahead logging (WAL) and periodic checkpoints.

A centralized database or shared storage service was considered as an alternative, since it would simplify recovery and provide a single authoritative state. However, this solution would introduce a stronger dependency on external infrastructure and partially contradict the decentralized nature of the system. The adopted design allows each node to recover independently after a crash, improving fault tolerance and resilience, at the cost of more complex local recovery logic.

Replication model: best-effort dissemination vs. reliable messaging The system propagates updates using best-effort inter-node replication combined with periodic anti-entropy synchronization.

An alternative would have been to rely on a reliable messaging middleware with acknowledgments, retries, and delivery guarantees. This option was not adopted in order to keep the architecture lightweight and avoid introducing additional infrastructural components. The chosen approach accepts that some updates may be temporarily missed, but guarantees that replicas eventually converge once anti-entropy has had enough time to reconcile their state.

Voting semantics: aggregated counters vs. individual ballots The system models votes as monotonic counter increments associated with poll options, rather than as individual user ballots.

A richer alternative would explicitly track voter identities and enforce constraints such as one vote per user, vote modification, or vote revocation. This was intentionally avoided because it would complicate the replicated data model and make conflict-free

merging more difficult. The selected approach is therefore better understood as distributed vote aggregation rather than as a full-featured election system.

Overall, the system prioritizes availability, fault tolerance, and conceptual simplicity over strict consistency guarantees and richer application semantics. These trade-offs are coherent with the project goal of demonstrating core distributed systems concepts in a realistic yet manageable prototype.

3.8 Fault-Tolerance

Fault tolerance is achieved through a combination of **replication**, **failure suspicion**, **retry-free eventual reconciliation**, and **crash recovery mechanisms**.

Replication. Each node stores a full logical replica of the poll state. Replication occurs in two complementary ways:

- **best-effort update propagation**: after processing a local vote, the node attempts to send the new counter update to all currently non-dead peers;
- **periodic anti-entropy synchronization**: nodes periodically retrieve full cluster state from peers and merge any missing or newer components.

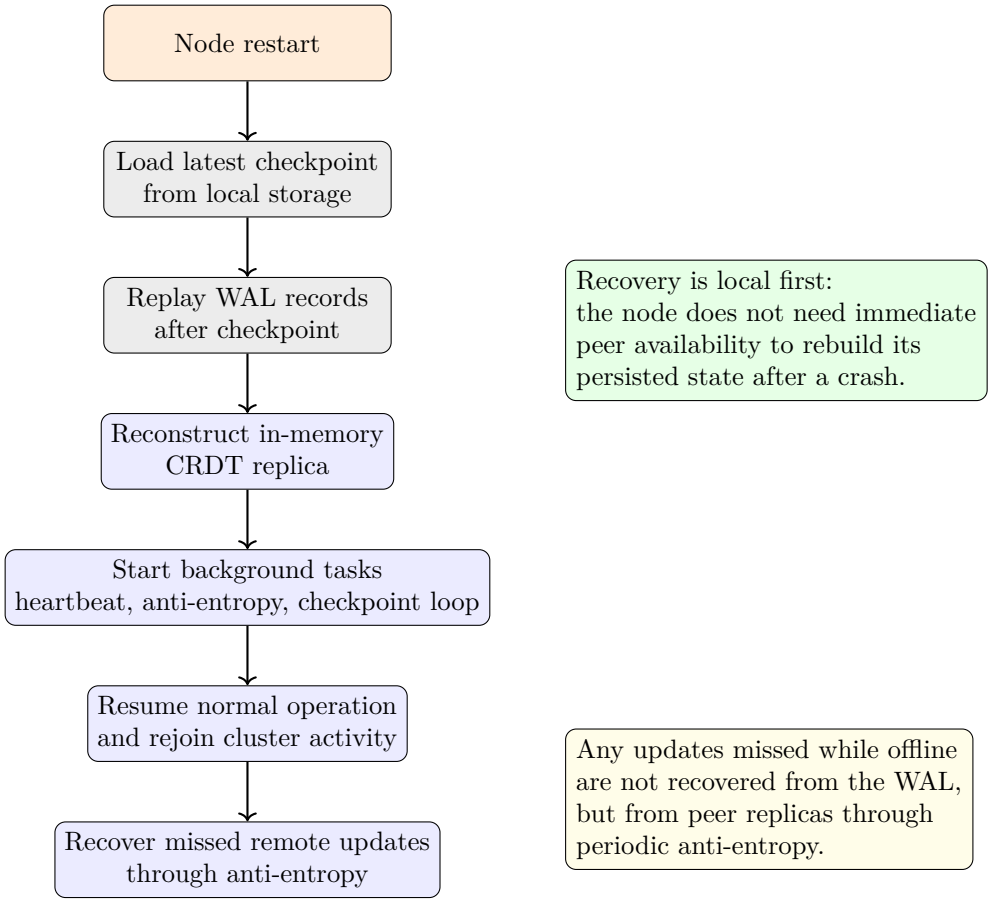
This combination allows the system to remain lightweight in normal conditions while still healing missed updates after temporary failures or partitions.

Heartbeats and timeouts. Each node periodically sends heartbeat messages to peers. Based on the time elapsed since the last successful heartbeat exchange, a peer is classified as:

- `UNKNOWN`: never seen since startup;
- `ALIVE`: recently reachable;
- `SUSPECT`: temporarily not heard from;
- `DEAD`: unreachable for a longer time.

This mechanism does not prove a crash with certainty; rather, it provides a practical suspicion-based estimate of peer liveness [1]. It is used to avoid trying immediate replication toward peers currently believed to be dead.

Error handling. Replication and heartbeat operations are handled in best-effort mode: failures are logged, but they do not cause the node to reject already accepted client requests. This is a deliberate design choice to preserve availability.


Node restart
Load latest checkpoint from local storage
Replay WAL records after checkpoint
Reconstruct in-memory CRDT replica
Start background tasks heartbeat, anti-entropy, checkpoint loop
Resume normal operation and rejoin cluster activity
Recover missed remote updates through anti-entropy
Recovery is local first: the node does not need immediate peer availability to rebuild its persisted state after a crash.
Any updates missed while offline are not recovered from the WAL, but from peer replicas through periodic anti-entropy.

Crash recovery. When a node crashes and later restarts, it reconstructs its state from the last checkpoint plus the remaining WAL records [2]. This allows recovery without depending on other peers, which improves resilience even when the restarted node is temporarily isolated.

3.9 Availability

The system prioritizes availability over immediate consistency.

Caching. No explicit caching layer is introduced, because each node already serves requests from its own in-memory replica. In practice, the local in-memory CRDT state acts as the working data structure for fast reads and writes.

Access path and proxying. Availability is achieved primarily by replication: any reachable node can process client requests independently. In addition, the deployment includes an Nginx reverse proxy that offers a single entry point for manual interaction with the cluster. This proxy does not introduce distributed coordination and does not implement sophisticated balancing policies; rather, it simplifies access by routing requests to specific nodes through stable paths such as `/node/1/`, `/node/2/`, and so on.

The proxy improves usability and makes the cluster easier to inspect manually, but it is not relied upon for correctness. The distributed guarantees still come from the replicated nodes themselves.

Direct access vs. proxied access. Two deployment modes are supported:

- a *default mode*, where only the proxy is exposed to the host;
- a *test/debug mode*, where node ports are also exposed directly.

The first mode provides a cleaner and more realistic single-entry-point setup. The second mode is useful for automated tests and controlled fault-injection scenarios, because it avoids transient proxy-side effects while nodes are being stopped and restarted.

Behaviour under network partitioning. In case of network partition, nodes continue to accept votes and answer queries using their local replicas. As a consequence:

- the system remains available on each reachable partition;
- replicas may temporarily diverge;
- once communication is restored, anti-entropy eventually reconciles state across nodes.

This behaviour directly reflects the target consistency model: temporary inconsistency is accepted in order to preserve service continuity [3, 5].

3.10 Security

Security is intentionally simplified in this project and is not among its primary goals. The system is designed mainly to explore replication, consistency, recovery, and fault tolerance rather than to provide production-grade protection mechanisms.

In particular, the prototype does not implement end-user authentication, fine-grained authorization policies, or transport-layer protection. It assumes a mostly trusted environment, while still distinguishing between public client-facing operations and internal node-to-node operations.

A minimal protection mechanism is adopted for internal endpoints, which are intended to be used only by peer nodes. In addition, incoming requests are validated to reject malformed data and preserve application-level integrity.

Overall, the security model should be understood as a lightweight prototype solution rather than a complete security architecture.

4 Implementation

This section reports the main technology-dependent choices adopted in the implementation of the distributed polling system.

4.1 Network protocols

The system uses **HTTP over TCP** for both client-to-node and node-to-node communication. This architectural choice was made to ensure interface uniformity and data monitoring transparency, providing an optimal balance between design simplicity and the ability to effectively verify network operations. Since every node exposes a REST API, the same communication model can be reused both for external clients and for internal replication among peers.

In particular, HTTP is used for:

- vote submission from clients to any reachable node
- poll listing and poll result retrieval
- best-effort dissemination of newly generated CRDT updates
- heartbeat exchange among nodes
- anti-entropy synchronization through full-state exchange and merge

4.2 Representation of in-transit data

All in-transit data is represented as **JSON**. JSON was chosen because it is human-readable, easy to serialize and deserialize in Python, and naturally supported by the adopted web framework.

The main exchanged payloads are:

- vote requests, containing a poll identifier and an option
- CRDT component updates, represented as tuples `(poll_id, option, node_id, value)`
- full replicated poll or cluster states used during anti-entropy
- heartbeat-related metadata and status information

The use of JSON also simplifies manual inspection of requests and responses during testing, which was particularly useful while validating replication, failure handling, and recovery behaviour.

4.3 Persistent state and querying

The system adopts a file-based persistence strategy instead of relying on an external database.

The application state is maintained in memory during execution, while durability is ensured through two complementary mechanisms:

- **Write-ahead log (WAL):** every update is appended to a log file before being applied to the in-memory state. This guarantees that no committed update is lost in case of a crash.
- **Checkpointing:** at regular intervals, the full state is serialized to a checkpoint file. This limits the size of the WAL and speeds up recovery.

Upon restart, a node reconstructs its state by loading the most recent checkpoint and replaying the updates stored in the WAL.

This approach provides a lightweight and transparent persistence mechanism, suitable for a prototype system where simplicity and portability are preferred over full database integration.

4.4 Authentication and authorization

The prototype does not implement end-user authentication. Any client able to reach the public API can submit votes and query poll data.

Authorization is likewise intentionally minimal. The system does not provide role-based or identity-based access control for external users, since all public operations are treated uniformly in the current project.

A distinction is instead enforced between public endpoints and internal endpoints. Internal node-to-node operations, such as replication and synchronization requests, are protected through a shared internal token transmitted in the request headers. This mechanism is not intended as a complete security solution, but rather as a lightweight safeguard preventing accidental or unauthorized access to internal coordination APIs during experiments.

This approach is coherent with the prototype nature of the project: security-sensitive features are simplified, while the implementation still preserves a basic separation between external usage and internal cluster communication.

4.5 Implementation remarks

The implementation follows the final design choice of using a **state-based CRDT approach** rather than timestamp-based conflict resolution [4]. Earlier design ideas considered logical timestamps for ordering distributed events, but the final prototype does not rely on timestamp arbitration. Instead, vote aggregation is represented through a **G-Counter-style replicated state**, where each node maintains its own monotonically increasing contribution for each poll option, and merge operations apply component-wise maxima.

This choice removes the need for conflict resolution based on total ordering and fits naturally with asynchronous best-effort replication and anti-entropy synchronization. As a consequence, replicas may temporarily diverge, but they eventually converge once communication is restored and synchronization has had enough time to run.

4.6 Technological details

The prototype is implemented in Python 3. The main technologies adopted are the following:

- **FastAPI** for the implementation of the REST API exposed by each node;
- **Uvicorn** as the Asynchronous Server Gateway Interface (ASGI) server used to run each node;
- **httpx** for asynchronous HTTP communication among nodes;
- **Pydantic** models for validation and serialization of request and replication payloads;
- **asyncio** background tasks for heartbeat emission, anti-entropy synchronization, and checkpoint scheduling;
- **Docker** for packaging each node into a containerized execution environment;
- **Docker Compose** for multi-node deployment and testing;
- **Nginx** as a reverse proxy, automatically configured to expose a single public entry point for manual access to the cluster;
- a small **HTML/CSS/JavaScript** frontend served by each node for simple interaction and visualization.

Each node is packaged through a dedicated Dockerfile based on a slim Python image. At runtime, environment variables are used to configure node identity, listening port, peer list, heartbeat timeouts, anti-entropy interval, checkpoint interval, storage location, and internal authentication token.

To simplify experimentation with clusters of different sizes, the project includes a Python utility that automatically generates:

- a Docker Compose configuration from the required number of nodes;
- an Nginx configuration exposing one proxied path per node.

The deployment utility also supports two operational modes: a default proxy-only mode for cleaner manual usage, and an `--expose-nodes` mode where node ports are published directly for automated validation and failure-oriented tests.

Overall, the chosen implementation stack privileges clarity, reproducibility, and ease of experimentation, which is appropriate for a prototype aimed at demonstrating replication, eventual consistency, crash recovery, failure detection, and deployment automation in a distributed system.

5 Validation

The validation strategy focused primarily on **integration testing**, **end-to-end testing**, and **repeated automated execution in Continuous Integration (CI)** rather than on extensive isolated unit testing. This choice was deliberate: the most relevant correctness properties of the project emerge from the interaction among multiple replicas, asynchronous communication, failure handling, and crash recovery, rather than from purely local computations. Therefore, the validation effort concentrated on distributed behaviour in realistic execution conditions, including temporary node failures, restart, divergence, convergence, and persistence. These aspects are directly related to the main project requirements, especially availability, partition tolerance, eventual consistency, durability, fault recovery, and concurrency support.

5.1 Automatic Testing

Unit testing. The project does not include an extensive suite of traditional isolated unit tests. The main reason is that the most critical properties of the system are not purely local algorithmic details, but distributed-system behaviours that only become observable when multiple nodes interact through the network.

Nevertheless, several internal mechanisms were indirectly validated through integration scenarios, including:

- monotonic CRDT counter growth,
- merge correctness and eventual convergence,
- write-ahead logging,

- checkpoint-based recovery,
- heartbeat-based failure suspicion.

Thus, instead of testing these mechanisms only in isolation, the validation strategy preferred to test them in the context in which they are actually used by the running distributed system.

Integration and communication testing. Communication and interaction among components were tested through an automated **PowerShell-based test suite**. These tests interact with the real REST API exposed by the nodes and exercise node-to-node communication under realistic conditions.

The suite verifies:

- client-to-node vote submission through HTTP,
- asynchronous replication of updates across nodes,
- anti-entropy synchronization after temporary divergence,
- heartbeat-based failure detection,
- crash recovery from checkpoint and WAL,
- idempotent handling of duplicated internal updates,
- convergence under concurrent writes,
- divergence and later healing after temporary disconnection.

End-to-end testing environment. The system is end-to-end tested in a **production-like containerized environment composed of three nodes**. The cluster is deployed through Docker Compose, and the automated validation is executed against real containers connected through the same virtual network used during normal experimentation.

This approach was preferred over a mocked environment because it validates the behaviour of the complete system stack:

- container startup and readiness,
- HTTP communication among nodes,
- state persistence across restarts,
- node stop/restart behaviour,
- anti-entropy and heartbeat background tasks,
- recovery after temporary failures.

Thus, deployment automation is also part of the validation strategy, since it allows the system to be repeatedly tested in a realistic distributed setting. This is particularly important for a project whose main properties concern replication, eventual convergence, and resilience to failures.

How to run the automated validation. All automated validation scenarios can be executed locally through the global PowerShell runner:

```
.\test\run_all.ps1
```

The runner starts from a clean environment, deploys the three-node containerized cluster, executes the validation scenarios, and then tears the environment down again. This improves reproducibility and reduces the risk that residual state from previous executions affects the results.

Repeated automated validation in CI. In addition to local execution of the automated suite, the project also includes a **GitHub Actions workflow** for repeated stress-style validation. The workflow is triggered on pushes, pull requests, and manual execution, and runs on an Ubuntu runner with Docker, Docker Compose, Python, and PowerShell available.

Its purpose is not to introduce new test scenarios, but to repeatedly execute the existing automated suite in a clean environment in order to detect intermittent failures, startup races, timing-related issues, and other forms of flakiness that are common in distributed systems.

More precisely, the workflow:

- creates a fresh `.env` file for internal authentication,
- generates a three-node Docker Compose configuration,
- deploys the distributed system from scratch,
- runs the full PowerShell test suite,
- collects logs for each execution,
- tears the environment down completely,
- repeats the entire process multiple times.

In the current version, the workflow repeats the full automated validation **20 times**. This repeated execution increases confidence in the robustness of the prototype, especially for properties depending on asynchronous propagation, anti-entropy, restart timing, and failure detection. If any run fails, the workflow reports the failure and uploads the collected logs as CI artifacts for later inspection.

Automated test scenarios.

Basic replication. **Rationale.** This scenario checks that updates submitted to different nodes are eventually replicated and that all replicas expose the same aggregated result.

How it was automated. Votes are submitted to different nodes through the public REST API, propagation is allowed to take place, and then all replicas are queried.

Requirements covered. FR2 (vote submission), FR3 (poll result retrieval), FR5 (distributed operation), FR6 (eventual propagation), NFR3 (eventual consistency).

Eventual consistency after node restart. **Rationale.** This scenario validates that a node missing updates while offline can later recover convergence after restart and anti-entropy synchronization.

How it was automated. One node is stopped, votes are submitted while it is unavailable, then the node is restarted and all replicas are polled until they expose the same state.

Requirements covered. FR6, NFR2 (partition tolerance), NFR3 (eventual consistency), NFR5 (fault recovery).

Failure detection. **Rationale.** This scenario checks that heartbeat-based monitoring detects when a peer becomes unreachable and later recognizes it as alive again after restart.

How it was automated. A node is stopped, another node is queried through the status endpoint until the peer is considered unavailable, and after restart it is expected to become reachable again.

Requirements covered. NFR1 (availability), NFR5 (fault recovery).

Crash recovery and persistence. **Rationale.** This scenario verifies that persisted state survives a restart and can be recovered locally.

How it was automated. Votes are submitted, persistence is allowed to complete, a node is restarted, and the test verifies that previously stored state is still available after recovery.

Requirements covered. NFR4 (durability), NFR5 (fault recovery).

Idempotent duplicated internal update. **Rationale.** Since distributed systems may deliver duplicate messages, internal update handling must be idempotent.

How it was automated. A valid internal replicated update is replayed more than once, and the final state is checked to confirm that duplicated delivery does not alter the result incorrectly.

Requirements covered. NFR3 (eventual consistency), NFR6 (concurrency support).

Concurrent updates convergence. **Rationale.** This scenario checks that concurrent votes submitted to multiple nodes do not cause conflicts or lost updates and that replicas eventually converge.

How it was automated. Concurrent requests are sent to different nodes, after which all replicas are queried until they converge to the expected counts.

Requirements covered. FR2, FR6, NFR3, NFR6.

Temporary disconnection and healing. Rationale. This scenario checks that the system remains operational while one node is temporarily disconnected, that replicas may diverge, and that they later converge again once communication is restored.

How it was automated. One node is temporarily isolated from the others, votes are submitted while it is disconnected, temporary divergence is observed, and then normal connectivity is restored so that anti-entropy can re-establish convergence.

Requirements covered. NFR1, NFR2, NFR3, FR6.

5.2 Acceptance test

In addition to automatic tests, a number of manual acceptance tests were performed.

What was tested manually. Manual testing focused mainly on:

- exploratory interaction with the web interface exposed by the nodes;
- access through the reverse proxy entry point;
- direct use of the REST API through browser or command-line tools;
- visual inspection of poll listing, vote submission, and result retrieval;
- observation of node status changes during stop/restart scenarios;
- qualitative verification that the system remains usable while replicas are temporarily diverging.

Why these tests were manual. These aspects were tested manually because they concern usability, visual behaviour, and exploratory observation of system behaviour. In particular, the browser-based interface and the proxy-based access path are more naturally assessed by a human tester than by a purely automated script.

Manual acceptance scenarios. The main manual scenarios were:

- opening the cluster through the reverse proxy and accessing individual node UIs;
- submitting votes through the web interface and verifying that results appear correct locally;
- querying different nodes while the cluster is healthy and confirming that results are eventually aligned;
- stopping one node and verifying that other nodes continue to accept votes and answer queries;

- restarting the stopped node and observing that it later rejoins the converged cluster state;
- inspecting the status endpoint to observe peer liveness information during failures and recovery;
- repeating selected checks in direct-node mode to validate behaviour without the proxy in the loop.

Overall, the combination of automated distributed tests, repeated CI execution, and manual acceptance checks provided reasonable confidence that the implemented prototype satisfies its main functional and non-functional requirements, especially those concerning distributed operation, eventual convergence, failure handling, crash recovery, and operational accessibility.

6 Deployment

The project is designed to be deployed through **containerization**, using Docker and Docker Compose. This choice simplifies execution, improves reproducibility, and allows the system to be instantiated as a multi-node cluster without requiring manual installation of dependencies on each node.

6.1 Software requirements

To run the project from scratch, the following software should be installed on the host machine:

- **Docker**, with support for container image building and execution;
- **Docker Compose**, used to deploy and manage the multi-node cluster;
- **Python 3**, used only to generate and stop the Docker Compose deployment automatically.

No manual installation of Python libraries for the application itself is required on the host system, since all node-level dependencies are installed inside the Docker images during the build phase.

The exact versions of Docker, Docker Compose, and Python are not strictly pinned in the repository, but a recent environment compatible with modern Docker Compose and Python 3 is required.

6.2 Configuration files and environment variables

The deployment relies on a `.env` file located in the project root. At minimum, the file should define the internal authentication token used for node-to-node communication. A minimal example is:

```
INTERNAL_TOKEN=my-token
```

This value is injected into each node container through Docker Compose and is used to protect internal endpoints reserved for replication, anti-entropy, and heartbeat communication.

In addition to the shared `.env` file, other configuration parameters are injected automatically as container environment variables when the deployment file is generated. These include:

- node identity (`NODE_ID`),
- listening port (`PORT`),
- peer list (`PEERS`),
- cluster size,
- persistent storage directory inside the container.

Timing-related parameters such as heartbeat intervals, failure-detection timeouts, anti-entropy interval, and checkpoint interval are not directly configured through environment variables. Instead, they are derived automatically at application startup based on the cluster size. This approach reduces configuration complexity while still allowing the system to adapt its behaviour to different deployment scales.

The deployment script also generates an `nginx.conf` file. This file contains the reverse-proxy configuration mapping public paths such as `/node/1/` to the corresponding node containers. Both the Docker Compose file and the Nginx configuration are derived automatically from the same requested cluster size, so they do not need to be edited manually by the user.

6.3 How to deploy the system from scratch

The recommended deployment procedure is the following.

Step 1 — Clone the repository. Obtain the project source code and move to the project root directory.

Step 2 — Create the `.env` file. Create a file named `.env` in the project root, containing at least:

```
INTERNAL_TOKEN=my-token
```

Step 3 — Generate and start the cluster. Run:

```
python run_cluster.py 3
```

This command generates:

- a file named `docker-compose.generated.yml`;
- a file named `nginx.conf`;

then builds the node image and starts a three-node cluster through Docker Compose.
The number `3` can be replaced with any desired cluster size.

Optional test/debug mode. If direct host access to node ports is also needed, run:

```
python run_cluster.py 3 --expose-nodes
```

This mode is useful for automated tests and failure-oriented inspection.

Expected outcome. After successful startup:

- a generated Docker Compose file is present in the project root;
- a generated Nginx configuration file is present in the project root;
- one container is created for each node;
- one proxy container is created;
- each node has a dedicated persistent Docker volume for WAL and checkpoints.

With the default numbering scheme adopted by the generator, node i listens internally on port $8000 + i$.

Step 4 — Stop and clean up the cluster. To stop the deployment and remove containers, networks, and persistent volumes, run:

```
python stop_cluster.py
```

6.4 Engineering of the deployment scripts

The deployment is not based on a static handwritten `docker-compose.yml` file. Instead, the project includes a Python script, `run_cluster.py`, that generates a deployment descriptor dynamically according to the requested number of nodes.
For each node, the generator creates:

- a distinct service name (e.g., node1, node2, node3);
- a node-specific identifier;

- an internal port;
- a peer list containing all other nodes;
- a dedicated Docker volume for persistent state;
- the cluster size, used by the application to derive timing-related parameters such as heartbeat intervals, anti-entropy scheduling, and failure-detection timeouts.

In addition, the generator always creates:

- a `proxy` service based on Nginx;
- an `nginx.conf` file exposing one public path per node.

The script supports two deployment variants:

- **default mode**, where only the proxy is published on the host;
- **exposed-node mode**, enabled with `--expose-nodes`, where node ports are also published individually.

This approach has several advantages. First, it avoids maintaining multiple deployment files for different cluster sizes. Second, it guarantees consistency between cluster topology, peer configuration, and proxy routing, since all of them are generated from the same source of truth. Third, it supports both human-oriented and test-oriented execution without duplicating deployment logic.

The generated services are built from the application image located in the `node` directory, and each node service mounts a dedicated persistent volume at `/data`. This design ensures that write-ahead logs and checkpoints survive container restart, which is essential for validating crash recovery behaviour.

The companion script `stop_cluster.py` complements this mechanism by shutting down the generated deployment through:

- container removal,
- network removal,
- volume removal,
- orphan cleanup.

As a consequence, the deployment process is fully scriptable, reproducible, and aligned with the current cluster topology.

6.5 Deployment remarks

The chosen deployment strategy prioritizes simplicity, reproducibility, and ease of experimentation over production-grade orchestration.

Using Docker Compose is appropriate for this project because it allows the system to:

- instantiate multiple interacting replicas easily;
- attach persistent volumes to each node;
- reproduce controlled failure and restart scenarios;
- expose either a single proxy entry point or individual node ports depending on the validation needs.

The introduction of the reverse proxy improves the engineering quality of the deployment compared to a purely direct-port approach. In particular, it offers a cleaner manual access path, reduces host-level port clutter in the default mode, and more closely resembles the presence of an ingress component in a more realistic deployment.

7 User Guide

This section explains how to use the distributed voting system once the cluster is running.

7.1 Starting the system

To start the system, follow the deployment procedure described in Section 6. Once the cluster is running, it can be accessed through the interfaces described below.

7.2 Recommended access path

The recommended manual access path is through the reverse proxy, which provides a single entry point to the cluster.

Open in a browser:

```
http://localhost:18080/
```

From there, specific node interfaces can be reached through paths such as:

```
http://localhost:18080/node/1/
http://localhost:18080/node/2/
http://localhost:18080/node/3/
```

If more nodes are started, the same numbering scheme continues:

```
http://localhost:18080/node/4/
http://localhost:18080/node/5/
...
```

If the cluster was started with `--expose-nodes`, each node can also be accessed directly through its host port, for example:

```
http://localhost:8001
http://localhost:8002
http://localhost:8003
```

Direct access is mainly useful for testing, debugging, and observing divergence and convergence across replicas.

7.3 What the web interface allows

From the web interface, a user can:

- submit a vote for a poll option;
- implicitly create a poll by casting the first vote for a new poll identifier;
- inspect the list of known polls;
- retrieve the current results for a poll;
- inspect local node status information.

7.4 Expected behaviour

The expected behaviour is the following:

- when a vote is submitted to one node, it is accepted locally immediately;
- the update is then propagated asynchronously to peer replicas;
- replicas may temporarily diverge;
- after sufficient synchronization time, all nodes eventually converge to the same aggregated results.

7.5 REST API examples

If the cluster has been started with direct node exposure, a vote can be submitted to node1 with:

```
Invoke-RestMethod -Uri "http://localhost:8001/vote" ‘
  -Method Post ‘
  -ContentType "application/json" ‘
  -Body ’{"poll_id":"poll1","option":"A"}’
```

Poll results can then be retrieved with:

```
Invoke-RestMethod -Uri "http://localhost:8001/poll/poll1"
```

Node status can be inspected with:

```
Invoke-RestMethod -Uri "http://localhost:8001/status"
```

7.6 Stopping the system

To stop and clean up the cluster, run:

```
python stop_cluster.py
```

8 Self-evaluation

8.1 Roberto Pisu

My role in the project covered the full development lifecycle: requirements analysis, architectural design, implementation, testing, deployment automation, and documentation. In particular, I was responsible for the design and implementation of the replicated voting logic, the REST-based node interface, the CRDT-based consistency mechanism, persistence through write-ahead logging and checkpointing, heartbeat-based failure detection, automated testing, and Docker-based deployment.

From a technical perspective, I consider the main strengths of the project to be the following.

- The project is strongly aligned with the distributed-systems topics covered in the course, especially replication, eventual consistency, fault tolerance, crash recovery, and distributed communication.
- The final design evolved in a meaningful way during development: an initial timestamp-based approach was replaced by a CRDT-based solution with anti-entropy synchronization, which proved conceptually cleaner and more robust under concurrent updates.
- The implementation goes beyond a minimal proof of concept, since it includes persistence, recovery, failure detection, automated validation, and deployment automation.
- The use of Docker and generated Compose configurations made the project reproducible and easier to validate under controlled multi-node scenarios.
- The presence of both local automated tests and repeated CI execution increased confidence in the correctness and robustness of the implementation.

Overall, I believe the project successfully translates core distributed-systems concepts into a coherent and testable implementation. Its main value lies in turning these ideas into an actual working system.

From a personal perspective, the project helped me better understand the practical challenges of engineering and validating asynchronous distributed behaviour.

9 Future works

The current prototype already demonstrates the main distributed-systems concepts targeted by the project, but several extensions would be needed to evolve it toward a more complete and production-oriented system.

- **Security improvements:** The security model could be strengthened through user authentication, authorization policies, and transport protection via HTTPS/TLS, which are intentionally simplified in the current prototype.
- **Validation and observability:** Validation and observability could be expanded through broader fault-injection scenarios, larger-scale stress tests, structured logging, metrics collection, and monitoring dashboards.
- **Improved replica coordination:** The current system relies on periodic anti-entropy and heartbeat-based failure suspicion. A possible extension would be to refine replica coordination through adaptive synchronization intervals, more selective state exchange, and more robust peer-management policies, in order to reduce unnecessary traffic and improve responsiveness under unstable network conditions.

References

[1] Tushar Deepak Chandra and Sam Toueg. "Unreliable Failure Detectors for Reliable Distributed Systems". In: *Journal of the ACM* 43.2 (1996), pp. 225–267. DOI: 10.1145/226643.226647.

[2] Elmootazbellah N. Elnozahy et al. "A Survey of Rollback-Recovery Protocols in Message-Passing Systems". In: *ACM Computing Surveys* 34.3 (2002), pp. 375–408. DOI: 10.1145/568522.568525.

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

[4] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*. Springer, 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29.

[5] Werner Vogels. "Eventually Consistent". In: *Communications of the ACM* 52.1 (2009), pp. 40–44. DOI: 10.1145/1435417.1435432.