Mini-Dynamo: A Distributed Key-Value Store in Python

Final Report for the Distributed Systems Course 2025/2026

Dominykas Arnauskas
dominykas.arnauskas@studio.unibo.it

1 Concept

This project delivers a **distributed key–value store** (a software service) and supporting tooling for demonstration and evaluation.

- **Product type:** a multi-node distributed storage service (application) plus a simple demo UI.
- **System goal:** accept client requests on any node and store/retrieve data reliably even when nodes fail.
- **Main functionalities:**
  - `PUT(key, value)`: store/update a value for a key.
  - `GET(key)`: retrieve the latest value for a key.
  - `DELETE(key)`: delete a key using a tombstone record.
  - Partitioning via consistent hashing (with virtual nodes).
  - Replication to R distinct nodes.
  - Quorum reads/writes via parameters W and Q.
  - Membership and failure detection via heartbeats.
  - Web UI for demo and observability (human-readable summaries + raw JSON).

2 Requirements Elicitation and Analysis

Functional requirements

- Provide a client API for `PUT/GET/DELETE`.
- Allow clients to contact any node (coordinator-style handling).

- Partition keys across nodes using consistent hashing.
- Replicate records to multiple nodes and support quorum-based reads/writes.
- Detect failed peers and adapt routing to any available nodes.
- Support changes when a known node becomes unreachable without requiring a full restart.
- Provide a simple user interface for demonstration.

Non-functional requirements

- Understandable implementation (clear modular structure; small components).
- Reproducible deployment (Docker + Docker Compose).
- Possibility for debugging (logging + a state endpoint).
- Unit tests for local logic, manual tests for multi-node behavior.

2.1 Relevant Distributed System Features

- **Distribution and scalability:** partitioning via consistent hashing.
- **Fault tolerance:** replication and continued operation when a node fails.
- **Consistency trade-offs:** quorum reads/writes with tunable parameters $R/W/Q$.
- **Partial failures:** timeouts, node unavailability, and retry behavior.
- **Membership:** detection of node liveness via heartbeats.

3 Design

3.1 Architecture

The system uses a **peer-to-peer cluster** where each node offers both public client endpoints and internal replica/membership endpoints. A client may send requests to any node; the receiving node acts as a **coordinator** for that request, computing the responsible replicas using the current hash ring and then contacting replicas to satisfy the requested quorum. This is done to avoid one permanent central coordinator.
Nodes expose:

- **Client API:** `/kv/put`, `/kv/get`, `/kv/delete`.
- **Internal replica API:** replica `put/get/delete` endpoints used for replication.
- **Debug/state API:** `/debug/state` for ring/membership visibility.
- **Membership API:** `/internal/heartbeat`.

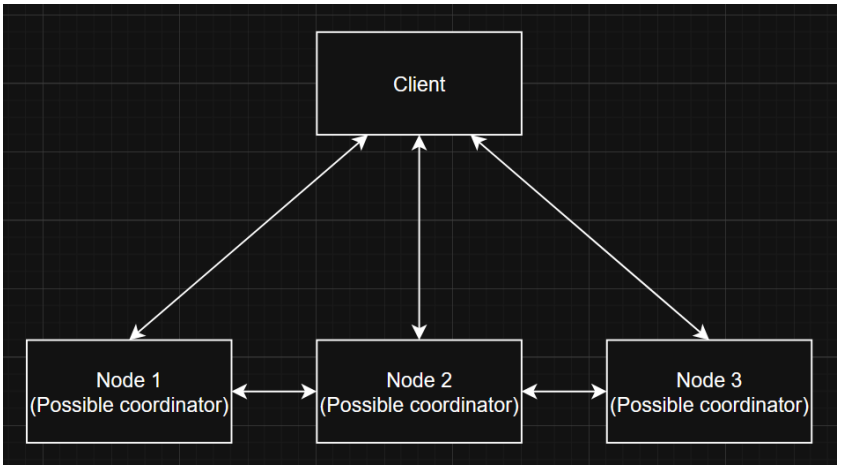

Figure 1: Simple architecture of Mini-Dynamo. A client can send a request to any node, and the receiving node acts as coordinator for that request.

3.2 Infrastructure

Nodes run as separate processes (or containers) and communicate over HTTP. Deployment uses Docker Compose to launch multiple nodes and the UI service on a shared virtual network. Ports are mapped to the host for easy access.

3.3 Modelling

Record model. Each key maps to a record containing:

- `value`: optional string (absent for tombstones),
- `ts`: a floating-point timestamp,
- `tombstone`: boolean deletion marker.

Conflict model. The system uses **last-write-wins (LWW)**: the record with the greatest timestamp is considered true. This was implemented due to it's simplicity.

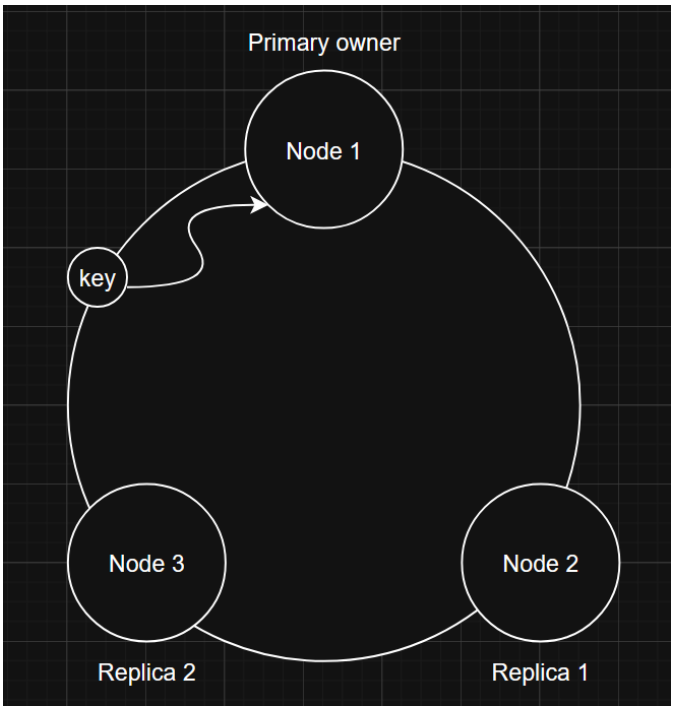

Figure 2: Example of key placement in the consistent hashing ring. The position hash(key) is mapped onto the ring, and the first node encountered clockwise becomes the primary owner. The following clockwise nodes are selected as replicas.

3.4 Interaction

Write path (PUT). A coordinator determines the replica set for the key (size R) and sends internal replica writes to those nodes. The operation is acknowledged to the client once W replicas confirm.
Read path (GET). A coordinator queries replicas and waits for Q responses, then returns the newest record via LWW.
Delete path (DELETE). A coordinator replicates a tombstone record with a timestamp and waits for W acknowledgments.

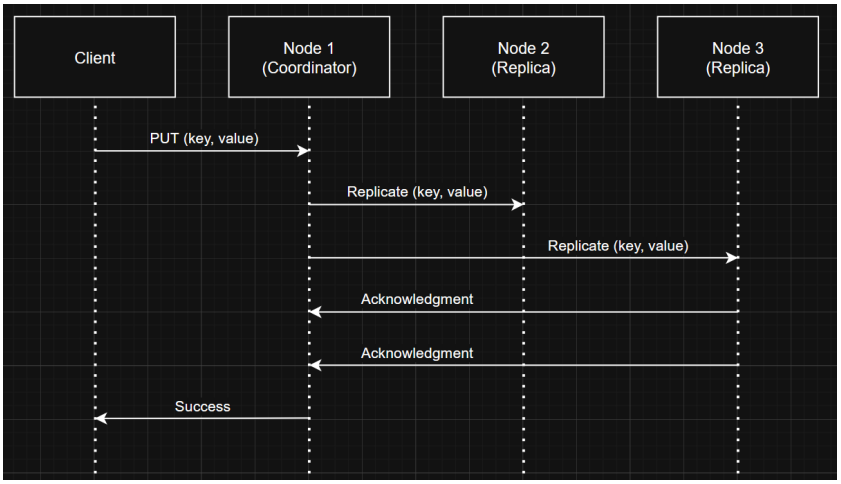

Figure 3: Sequence diagram for a PUT operation. The coordinator receives the client request, forwards replica writes, waits for acknowledgments, and then returns success.

3.5 Behaviour

The system behaves correctly under the following normal and failure scenarios:

- With no failures, reads and writes succeed and distribute load across nodes.
- If one node fails, the operations still succeed. This only happens if the required quorum can be satisfied by the remaining live replicas.

3.6 Data and Consistency Issues

Mini-Dynamo targets **eventual consistency**. With appropriate quorum choices, the probability of serving stale reads is reduced. Timestamps implement a simple conflict resolution scheme. A key observation during development was that weak write quorums (e.g., $W = 1$ with $R = 2$) can lead to data loss under a single-node failure if the only acknowledged replica fails.

3.7 Fault-Tolerance

Fault tolerance is achieved through replication and failure detection:

- **Replication:** data is stored on R distinct nodes.
- **Failure detection:** heartbeats mark peers as unavailable after a timeout.
- **Adaptive routing:** unavailable peers are excluded from the active ring.

3.8 Availability

Availability depends on quorum configuration and the number of live replicas. The system can be configured to favor availability (small W and Q) or stronger consistency/durability (larger W and Q).

3.9 Security

Security was not a primary focus for this prototype. The system assumes a trusted network environment (e.g., Docker Compose network) and does not implement authentication, authorization, TLS, or input hardening beyond basic validation. These could be added as future work.

4 Implementation

4.1 Technological details

- **Language:** Python.
- **Web framework:** FastAPI for node APIs and the demo UI.
- **Inter-node/client requests:** `httpx` (async HTTP client).
- **Concurrency:** `asyncio` for concurrent requests and background tasks (heartbeats).
- **Deployment:** Docker and Docker Compose.
- **Testing:** `pytest` for unit tests (hashing/store logic) and manual tests for multi-node behavior.

The codebase is organized into modules (hashing, membership, quorum logic, storage, node API, and UI) to keep the implementation navigable and to support easy debugging.

5 Validation

5.1 Automatic Testing

Two automatic tests were implemented:

- **Hashing tests:** validate consistent hashing properties (stable ownership, unique replicas).
- **Storage tests:** validation of storage semantics (LWW conflict resolution, tombstones).

5.2 Acceptance test

A manual acceptance test is performed using the web UI:

1. Start a 3-node cluster (Docker Compose).
2. Use the UI to PUT a key and then GET it from the cluster.
3. Stop one node and repeat GET/PUT operations to demonstrate fault tolerance.

6 Deployment

Deployment is provided via Docker and Docker Compose. A single Docker image is built for the project, and Compose runs multiple node services plus the UI service. This enables reproducible demonstrations on any machine with Docker installed.

7 User Guide

Local run (without Docker). Start three nodes in separate terminals using `python run_node.py` with different ports and peer lists, then start the UI using `python run_ui.py`.
Docker run. From the project root:

```
docker compose up --build
```

Open the UI at `http://127.0.0.1:9000`. The UI supports PUT/GET/DELETE and shows a simplified view of the node state (active ring, alive/dead peers, and $R/W/Q$ configuration).

8 Self-evaluation

This project met the main learning objectives: implementing a realistic data-partitioning scheme, adding replication, exploring quorum-based trade-offs, and handling partial failures via timeouts and membership updates. The most instructive aspect was observing how configuration choices (especially W) affect durability under failures.

9 Future works

Possible extensions include:

- **Persistence:** store records permanently (e.g., SQLite or append-only log) to survive full restarts.
- **Read repair:** after a read, push the newest record to stale replicas.
- **Hinted handoff:** temporarily store writes for unavailable nodes and forward them upon recovery.

- **Gossip membership:** replace centralized peer lists/heartbeats with gossip dissemination.
- **Rebalancing:** when a new node joins or leaves, an automatic key redistribution would occur to ensure no data loss.
- **Security:** authentication, TLS, and input hardening for untrusted networks.