Distributed Sensor Hub

Final Report for the Distributed Systems Course 2025/2026

Alex Santini
alex.santini2@studio.unibo.it

May 3, 2026

Distributed Sensor Hub is a fully decentralized peer-to-peer platform for smart-city sensor data collection and replication, designed to operate across multiple nodes without any central coordinator. Each node may manage connections to multiple local sensors, ingest their readings through a pluggable sensor interface to support different sensor types, and maintain a local replica of the shared system state.

The platform disseminates sensor updates through push-pull gossip and resolves conflicts with deterministic timestamp-based Last-Writer-Wins merging, providing eventual consistency across replicas.

To remain operational in realistic distributed environments, the system combines gossip-based membership, phi-accrual failure detection, and anti-entropy recovery after crashes or temporary network partitions. In CAP terms, the project prioritizes availability and partition tolerance over strong consistency, while still guaranteeing convergence once communication is restored.

Disclaimer

During the development of the project and the preparation of this report, the author used Large Language Models (LLMs) as auxiliary tools for limited coding support, brainstorming, and improving the clarity and structure of the written text.

All outputs produced with the assistance of LLMs were critically reviewed, validated, and, when necessary, revised by the author. All architectural decisions, system design choices, and implementation details were defined and verified by the author, who takes full responsibility for the final content of this report.

1 Concept

Product type. *Distributed Sensor Hub* is a decentralized peer-to-peer software platform for distributed sensor monitoring. It combines:

- a set of autonomous, containerizable peer nodes that collect local readings, synchronize state in a distributed cluster, and replicate data without a central coordinator;
- an HTTP API for observability and integration;
- a read-only web UI for monitoring.

Use case description. What is the software doing? Each node maintains active connections to its local sensors, continuously collects readings, and updates its own local state. In parallel, the node shares its sensor-state updates with peer nodes and receives their updates through a gossip-based push-pull protocol. As a result, every node progressively learns and replicates the state produced by other nodes, building a distributed shared view. Eventual consistency is guaranteed through timestamp-based conflict resolution, while the system remains operational under failures, delays, and temporary partitions.

Where are the users? In a smart-city setting, users are in two main groups. Technical operators/developers manage container deployment and node setup, and they extend the system because sensor protocols are injectable and node-to-node connection policies are pluggable; they may also build tools that consume exported data. Information-oriented users access the resulting data views to monitor conditions and support practical decisions or downstream services.

When and how frequently do they interact with the system? Interaction is role-dependent. Technical operators/developers interact mainly during setup, topology changes, protocol/policy development, testing, and incident handling; this is recurring but not continuous. Information-oriented users interact when they need situational updates, monitoring views, or derived outputs for operational decisions. In contrast, the distributed core operates continuously and autonomously: sensors generate readings periodically and nodes exchange synchronization updates continuously.

How do they interact with the system? Technical operators/developers interact through deployment/runtime tooling (local process management or Docker orchestration), protocol/policy implementation components, and programmatic inspection via HTTP endpoints. Information-oriented users interact through read-only monitoring views, typically via the web UI or external dashboards/tools that consume exported data.

Does the system need to store user data? Which? Where? No user-specific data is stored. The system handles technical runtime data only: timestamped sensor readings, replicated node state, synchronization metadata, and membership/peer information. Data is maintained locally by each node (in-memory in the current scope) and propagated across peers by the gossip protocol.

Multiple roles The model includes three technical roles:

- *Node (core role)*: ingests local sensor data, maintains local state, and participates in peer synchronization and replication;
- *Technical operator/developer*: deploys and configures nodes, develops injectable sensor protocols and pluggable connection policies, and validates runtime behavior;
- *Information-oriented user/client*: consumes read-only state views for monitoring and downstream operational use, without participating in replication.

Why distribution is needed. Distribution is a core requirement, not an implementation detail. In a smart-city environment, sensing points are geographically distributed across different physical areas, so data is naturally produced at multiple nodes rather than at a single central site. Each node contributes local observations and shares them with peers, allowing the system to build a replicated global view without relying on a permanent coordinator.

Distribution is also required for fault tolerance. Nodes share sensor state, membership knowledge, topology information, and replication metadata through peer-to-peer communication, while remaining able to operate autonomously if other peers become unreachable. This allows the system to continue local ingestion and observation during node crashes, delays, or temporary network partitions, and to converge again when communication is restored.

Finally, distribution is essential because the project is meant to expose real distributed-systems behavior, including delay, partial failure, recovery, and eventual consistency. These aspects are part of the project goals and must be observable rather than hidden behind a centralized architecture.

The detailed behavioral requirements, protocol constraints, and acceptance criteria are specified in section 2.

2 Requirements Elicitation and Analysis

This section provides functional, non-functional, and implementation requirements for the analysed system, focusing on distributed sensor ingestion, replicated-state convergence, fault observability, and reproducible deployment.

2.1 Relevant Distributed System Features

Not all distributed-system features have the same priority in this project. The primary ones are decentralization, eventual convergence, partition/failure tolerance, and observability. Production-grade security and very-large-scale elasticity are outside the current scope.

Distribution transparency. Partially relevant. Low-level details should be abstracted for clients, but distributed behavior must remain visible. Delays, membership evolution, liveness changes, and recovery are intentionally observable via API and dashboard.

Decentralization. Highly relevant. Each node runs the full runtime stack independently. Normal operation must not depend on a permanent master, central broker, or external shared database.

Fault tolerance and recoverability. Highly relevant. Surviving nodes must continue local operation during crashes or partitions, and reconnected nodes must recover missing state from reachable peers (delta sync when possible, fuller transfer when needed). Recovery is eventual and parameter-dependent, not real-time guaranteed.

Failure detection and fault observability. Highly relevant. Liveness is inferred from heartbeat evidence and exposed as states such as `alive`, `suspected`, and `dead`, together with events and replication metrics.

Consistency. Highly relevant. The model is eventual consistency with deterministic conflict resolution: replicas may diverge temporarily but converge after exchanging the same updates. Strong immediate consistency is not a goal.

Partition tolerance. Relevant. During temporary partitions, nodes continue local ingestion/observation while replicas may diverge; after healing, synchronization reconciles reachable replicas.

Scalability. Relevant at intended scale. With full-mesh connectivity, the system targets small/medium clusters; alternative injectable topology policies can support larger deployments.

Resource sharing. Highly relevant. Nodes share distributed knowledge (sensor winners, membership/liveness data, topology metadata, events, metrics) through message exchange and deterministic merge rules, not through a central store.

Openness, interoperability, and heterogeneity. Relevant. The architecture separates sensor providers, topology policies, runtime/protocol logic, replicated state, and observation interfaces, so sensor protocols and connection policies remain replaceable.

Interoperability is provided through shared read-only HTTP surfaces for dashboards, scripts, and tools.

Evolvability and maintainability. Highly relevant. The implementation should remain modular (runtime, networking, protocol, membership, replication, state, sensors, topology, API/UI) to support debugging, validation, and extension.

Performance, concurrency, and communication efficiency. Relevant, without hard real-time targets. Nodes must support concurrent sensing, membership, replication, and API serving; timing and sync parameters should stay configurable for stable behavior. No strict latency/throughput SLA is required.

Security and trust. Limited in current scope. The system assumes trusted environments and does not currently provide authentication, authorization, or transport encryption. Security hardening is a future extension.

Economy and cost. Relevant as a constraint. The project should run on commodity machines using repository artifacts and Docker Compose, without paid managed infrastructure. Cost efficiency is secondary to dependable distributed behavior.

2.2 Functional Requirements

FR01 – Multi-node autonomous operation. The system shall allow multiple nodes to start independently and participate in the same distributed deployment without requiring a central coordinator.

Acceptance criterion. When a cluster of at least two configured nodes is started, each node becomes operational as an autonomous process and exposes its local service interfaces without depending on a single master node.

FR02 – Local sensor ingestion. Each node shall be able to manage one or more local sensor sources or sensor connections and ingest their readings into the distributed system state.

Acceptance criterion. Given at least one configured sensor on a node, the node eventually exposes fresh sensor records associated with its own origin through the read API or introspection API.

FR03 – Cluster-wide sensor-state dissemination and synchronization. The system shall propagate sensor updates among nodes through decentralized peer-to-peer replication with bidirectional synchronization so that readings produced on one node contribute to a shared replicated view and become visible on other reachable nodes.

Acceptance criterion. If a node generates new sensor readings while communication paths to other nodes are available, those readings eventually appear in the replicated sensor-state view exposed by the other reachable nodes. Reachable peers exchange updates in both directions through push/pull synchronization, so each node can advertise local changes and retrieve missing records.

FR04 – Deterministic timestamp-based conflict resolution. The system shall resolve concurrent or conflicting updates through a deterministic Last-Write-Wins merge

policy so that replicas can converge to the same winning value for each logical sensor record.

Acceptance criterion. When two or more updates compete for the same logical sensor record, all nodes that receive the same set of updates eventually expose the same winner for that record.

FR05 – Decentralized membership and liveness observation. The system shall allow nodes to discover peers through bootstrap and peer-to-peer membership exchange, maintain a distributed view of cluster membership, and provide operators with a node-local assessment of peer liveness.

Acceptance criterion. After startup and sufficient message exchange, a node exposes a membership view containing the peers it has discovered directly or indirectly; if liveness evidence from a known peer degrades or stops for long enough, at least one observing node eventually reports that peer as `suspected` or `dead` in its membership/introspection output.

FR06 – Recovery and rejoin after failures or partitions. The system shall allow a node that restarts, reconnects, or rejoins after temporary unavailability to recover the replicated state needed to resume normal operation by synchronizing with reachable peers, including cases where incremental synchronization alone is insufficient.

Acceptance criterion. After a node crash, temporary isolation, or long partition, once communication with at least one sufficiently up-to-date peer is restored, the recovering node eventually exposes a replicated state aligned with the rest of the reachable cluster, using a more complete recovery path when incremental synchronization is not enough.

FR07 – Read-only observability interface. The system shall provide a read-only programmatic interface for observing sensor state, membership, topology-related information, events, and operational metrics.

Acceptance criterion. A client can query documented HTTP `GET` endpoints and obtain structured snapshots of the node's current distributed-system view without modifying runtime state.

FR08 – Human-oriented monitoring support. The system shall support human observation of the running cluster through a dashboard or equivalent visual monitoring interface.

Acceptance criterion. An observer can inspect live cluster information from a web browser through a dashboard that consumes the exposed read-only API of a node.

2.3 Non-Functional Requirements

NFR1 – Decentralization. Normal operation shall not depend on any central coordinator or single control component.

Acceptance criterion. The system continues to ingest local readings, maintain local state, and serve read-only observability data on surviving nodes even if one arbitrary peer becomes unavailable.

NFR2 – Eventual consistency. The replicated sensor state shall provide eventual, not immediate, consistency across nodes.

Acceptance criterion. In the absence of new conflicting updates and after communication is restored, reachable replicas converge to the same winning values for the same sensor records, yielding a consistent shared view across nodes.

NFR3 – Partition tolerance with degraded semantics. The system shall remain available for local operation during temporary network partitions, accepting that different partitions may temporarily diverge.

Acceptance criterion. During a partition, nodes in each connected component continue local ingestion and observation; after the partition heals, their replicas eventually converge again.

NFR4 – Fault observability. The system shall expose failures, liveness transitions, and recovery behavior clearly enough to support operational monitoring and debugging.

Acceptance criterion. Operators can inspect status changes, replication activity, and recovery evidence through exposed membership, event, or metric views.

NFR5 – Reproducibility of deployments and validation. The platform shall support repeatable deployments and validation scenarios on development machines without relying on external managed infrastructure.

Acceptance criterion. Using the documented setup and provided Docker Compose topologies, an operator can start a predefined cluster and reproduce representative convergence, partition, and recovery scenarios across repeated runs without relying on external managed services.

NFR6 – Extensibility of runtime policies and sensor sources. The system shall be open to additional sensor producers and alternative topology policies without changing the conceptual role of the rest of the distributed runtime.

Acceptance criterion. New sensor sources or alternative topology strategies can be introduced while preserving the same downstream ingestion, replication, membership, and observation capabilities.

NFR7 – Operational scalability at intended scale. The platform shall support small and medium peer deployments rather than only a fixed two-node demo.

Acceptance criterion. The documented deployment can be instantiated with multi-node topologies and provides replicated-state and observability functions across the deployment. Validation is required at least on six-node clusters; twelve-node clusters are supported as a deployment target.

2.4 Implementation Constraints

IR1 – Python-based implementation. The project shall be implemented in Python.

Justification. Python provides a practical balance of development speed, readability, and standard-library support for networking, concurrency, HTTP serving, serialization, and testing.

Acceptance criterion. The distributed node runtime, protocol handling, failure detection, replication logic, and observability services are implemented in Python and can be executed with the documented Python environment.

IR2 – Container-based multi-node deployment support. The project shall support multi-node deployment through Docker Compose.

Justification. Docker Compose enables reproducible multi-node topologies with consistent configuration, without requiring external orchestration platforms.

Acceptance criterion. An operator can launch a documented multi-node deployment using Docker Compose and obtain a running cluster with configured node identities, networking, and observability endpoints.

IR3 – Self-contained execution on commodity machines. The project shall be executable on commodity development machines without requiring paid cloud services, institution-managed infrastructure, or external managed services.

Justification. Self-contained execution keeps deployment low-cost, accessible, and reproducible directly from repository artifacts.

Acceptance criterion. An operator can start one or more nodes using the repository code, documented dependencies, and provided deployment artifacts without provisioning external infrastructure.

3 Design

This chapter explains the strategies used to meet the requirements identified during the analysis phase (section 2). It presents the system design choices that support the functional goals and the main non-functional constraints.

3.1 Architecture

The node architecture is organized as a **layered/modular design**. The goal is to keep transport, protocol semantics, domain logic, and observability concerns explicitly separated inside each runtime instance.

At node level, the internal structure is organized into explicit layers:

- `runtime`: component composition, lifecycle orchestration;
- `domain`: `membership`, `fd`, `gossip`, `state`, `topology`, `sensors`;
- `protocol`: message contracts, codec, dispatcher, handler routing;
- `observability`: `introspection` and `webapi`.
- `networking`: TCP transport, framing, connection management, retries;

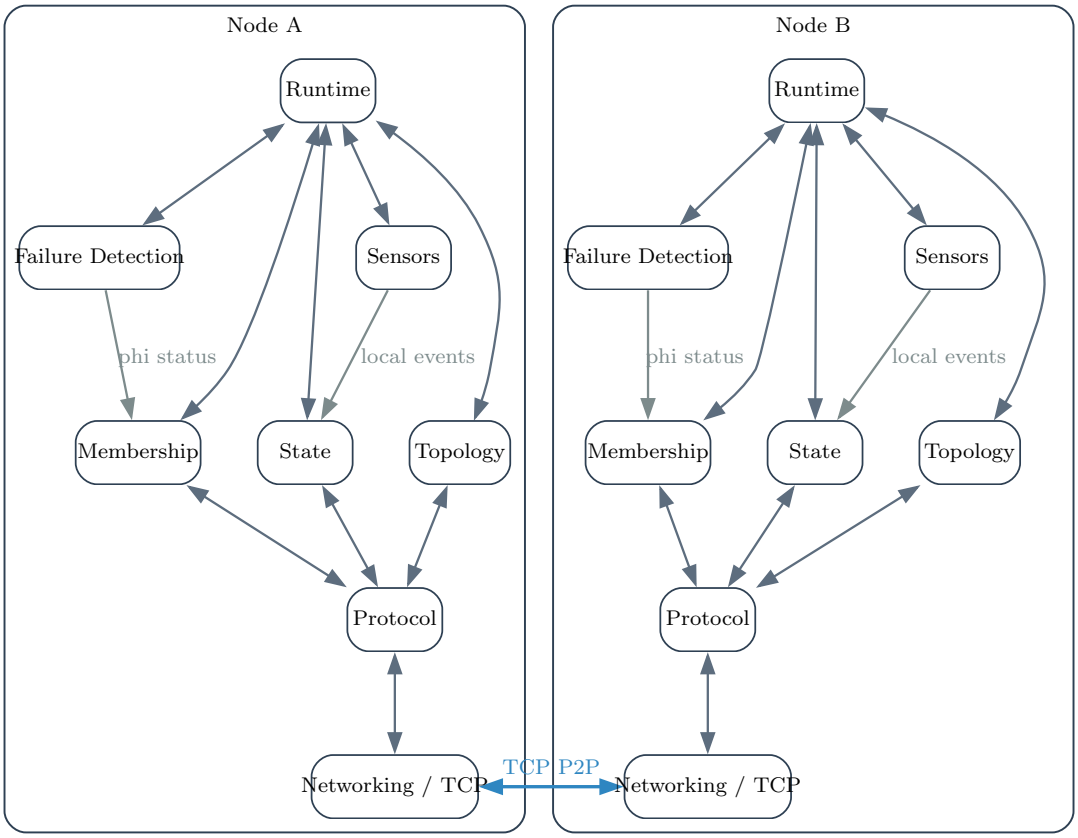

Figure 1: Internal layered architecture of a node runtime.

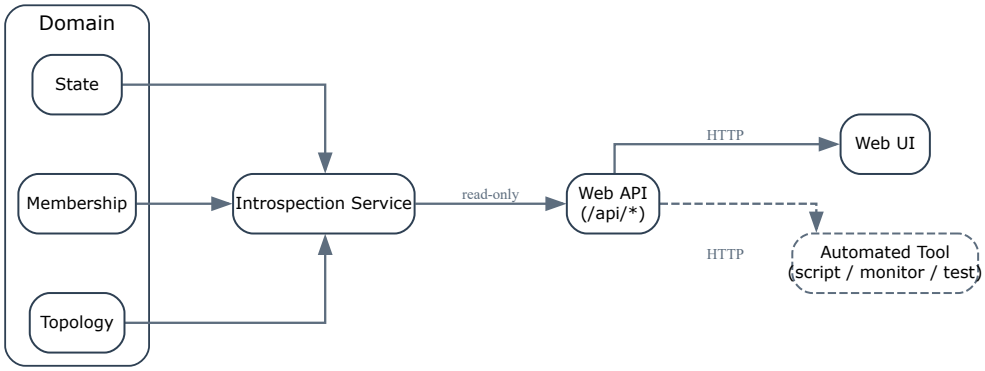

Figure 2: Observability architecture: introspection and read-only API exposure path.

This architecture separates transport concerns, protocol semantics, domain logic, and

observability concerns, improving maintainability and evolvability of the node runtime.

3.2 Infrastructure

The infrastructure is intentionally minimal and consists only of **peer nodes**. In the base deployment, the system runs with N homogeneous node processes. Each node embeds the same runtime stack (runtime, domain services, protocol handling, TCP networking, and observability endpoints), with no role specialization and no centralized infrastructure component.

Node distribution over the network follows a symmetric P2P model:

- all nodes are equivalent peers and communicate through bidirectional TCP connections;
- nodes may run on a single machine (development), within the same LAN/VPC, or across different networks and hosts;
- the only requirement is mutual reachability and TCP communication across configured inter-node endpoints.

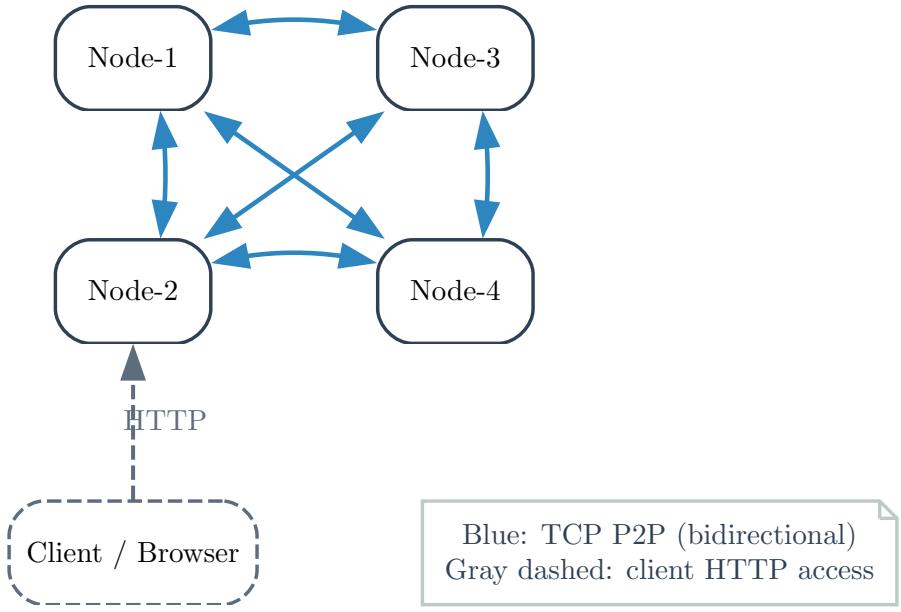

Figure 3: Infrastructure-level node connectivity in the base full-mesh deployment.

Node discovery is bootstrap-based and membership-driven:

- explicit node naming (`node-1`, `node-2`, ...) with known `host:port`;
- initial bootstrap through a static seed list in configuration;

- progressive discovery and cluster-view maintenance through membership/gossip exchange.

3.3 Modelling

At implementation level, the model is organized around node-local runtime components and their data-flow relationships.

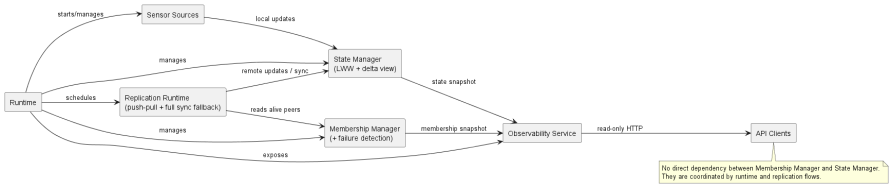

Figure 4: Domain model of node-local components and inter-component data flow.

The model includes the following components:

- **Runtime**: root orchestrator that manages the lifecycle of the internal subsystems;
- **Sensor Handler**: ingestion boundary for local sensor events;
- **State Store / Worker**: node-local replicated state authority and delta source;
- **State Replication Publisher**: component that reads replication deltas and propagates them to peers;
- **Peer Table**: local membership view of known peers and status;
- **Heartbeat Runtime**: periodic liveness loop and membership-update driver;
- **Heartbeat Monitor (ϕ)**: local failure-detection component owned by membership logic;
- **Topology Store**: local connectivity/topology view;
- **Introspection Service**: read-only aggregation endpoint;
- **API Clients**: external consumers accessing introspection over HTTP.

The main relationships represented in the model are:

- `Runtime` manages `Sensor Handler`, `State Replication Publisher`, `Heartbeat Runtime`, `Peer Table`, `State Store / Worker`, and `Topology Store`;
- `Sensor Handler` writes local sensor events to `State Store / Worker`;
- `State Replication Publisher` reads replication deltas from `State Store / Worker` and selects peers through `Peer Table`;

- `Heartbeat Runtime` evaluates/updates peer statuses in `Peer Table` and marks connect/disconnect information in `Topology Store`;
- `Peer Table` owns `Heartbeat Monitor` (ϕ);
- `State Store / Worker`, `Peer Table`, and `Topology Store` expose state/membership/topology snapshots to `Introspection Service`;
- `API Clients` consume `Introspection Service` through read-only HTTP.

This model makes explicit that authoritative operational knowledge remains node-local and in-memory, while inter-node convergence is achieved by replication and membership protocols.

3.4 Interaction

There are two main interaction families in the system.

The first is **peer-to-peer interaction among nodes**. This interaction is best-effort, message-based, and decoupled through protocol dispatch. It runs over TCP with client-side retry/queue behavior, but without end-to-end application-level delivery guarantees. It is used for:

- bootstrap and peer discovery;
- heartbeat exchange;
- dissemination of membership knowledge;
- dissemination and retrieval of replicated sensor state;
- recovery synchronization after missed updates.

The second is **observer interaction**, where external clients query a node through the HTTP read API. This interaction is request-response and read-only. In practice, the dashboard mainly relies on polling of `/api/introspection`, alongside other legacy read-only endpoints. It does not participate in cluster coordination and it never mutates the distributed state.

At the inter-node level, communication follows a layered path:

> *domain intent* $\rightarrow$ *protocol message* $\rightarrow$ *transport delivery* $\rightarrow$ *Message.decode* $\rightarrow$ *MessageDispatcher* $\rightarrow$ *handler* $\rightarrow$ *domain update*

The main interaction patterns are:

- **bootstrap and peer-list exchange**, used when a node joins and needs initial peer knowledge;
- **periodic heartbeat exchange**, used to collect liveness evidence;

- **gossip dissemination**, used mainly to spread membership and liveness knowledge without central coordination;
- **push-pull state synchronization**, used to exchange recent sensor-state updates efficiently;
- **full synchronization on demand**, used when incremental recovery is insufficient;
- **polling-based observation**, used by the dashboard and external tools to inspect cluster state.

In implementation terms, `JOIN_REQUEST`/`PEER_LIST` are event-driven (startup and peer discovery), `PING`/`PONG` and `GOSSIP_STATE` run at heartbeat cadence (`HEARTBEAT_INTERVAL_MS`), and replication rounds run at `GOSSIP_SYNC_INTERVAL_MS`. Push (`SENSOR_UPDATE`) and pull (`GET_DELTA`) target random subsets of peers currently marked `alive`, with fanout and pull cadence controlled by `GOSSIP_PUSH_*` and `GOSSIP_PULL_*` parameters. `FULL_SYNC_REQUEST` is used both as fallback after `DELTA_UNAVAILABLE` and during bootstrap to initialize state against seed peers. Topology has no dedicated request path: local connectivity/status changes update node-local topology state, which is then disseminated through the next gossip publications.

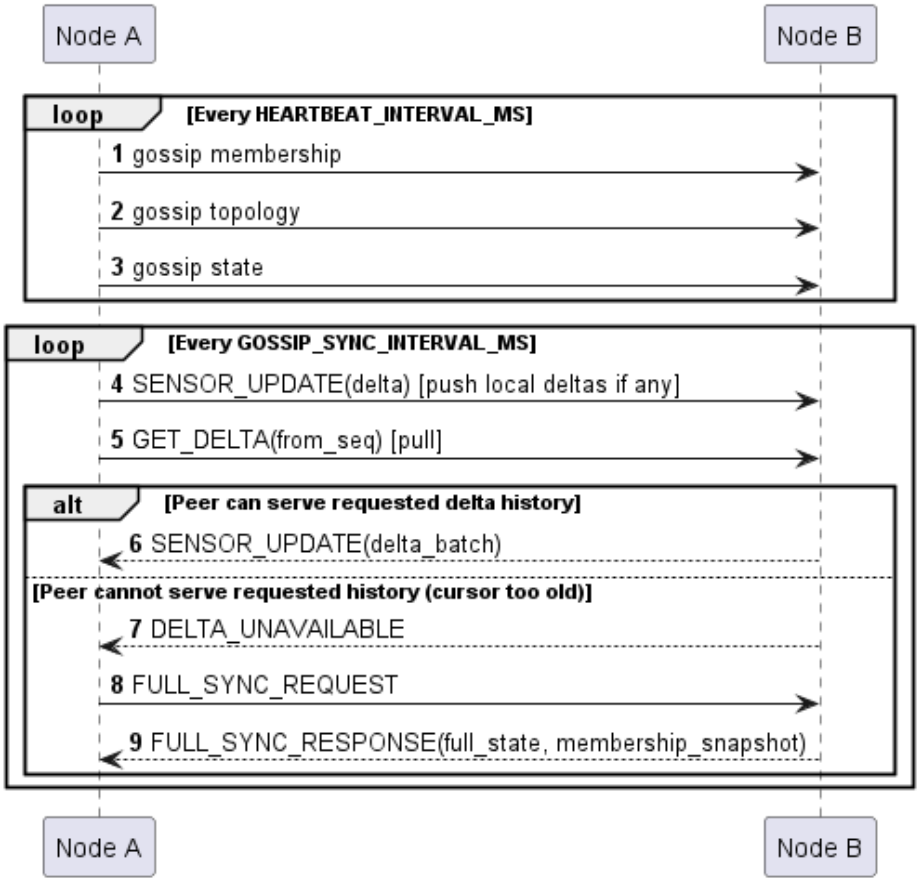

Figure 5: High-level overview of interaction and data propagation across peers and observer clients.

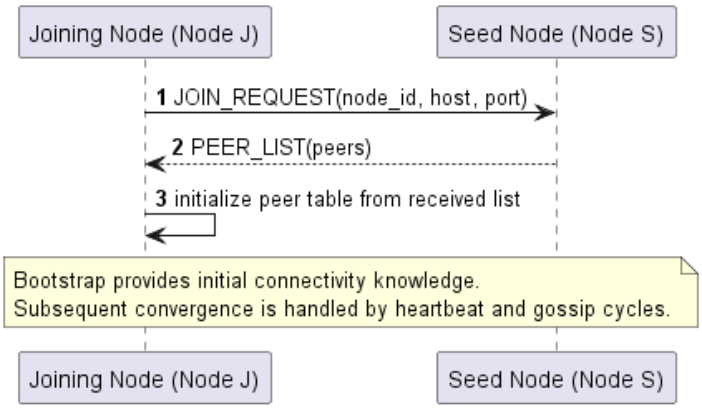

Figure 6: Bootstrap and peer-list exchange interaction for node join.

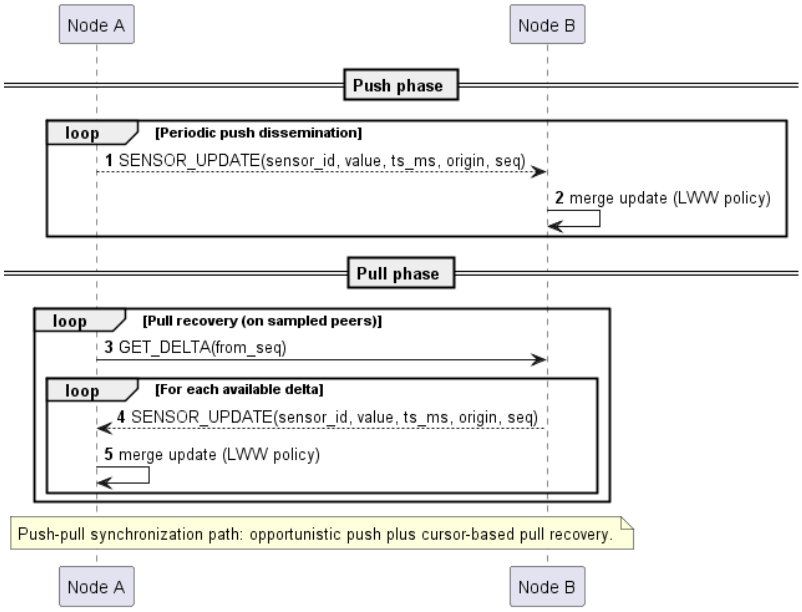

Figure 7: Incremental delta synchronization interaction.

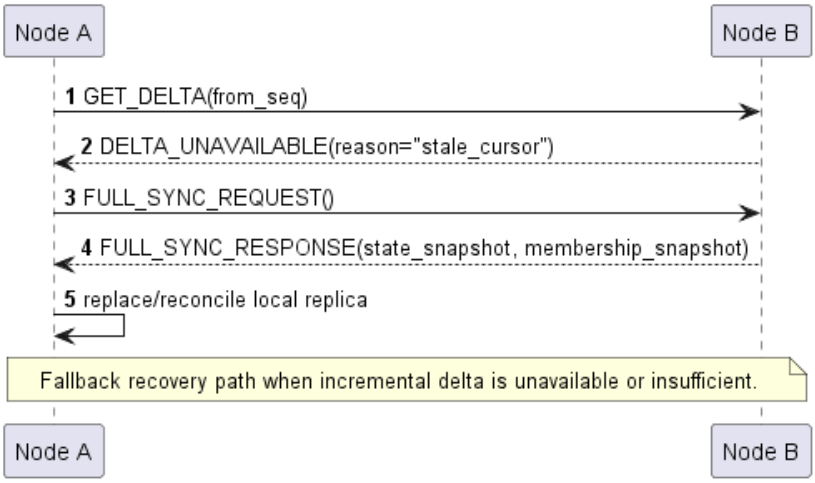

Figure 8: Full synchronization fallback when incremental delta is unavailable.

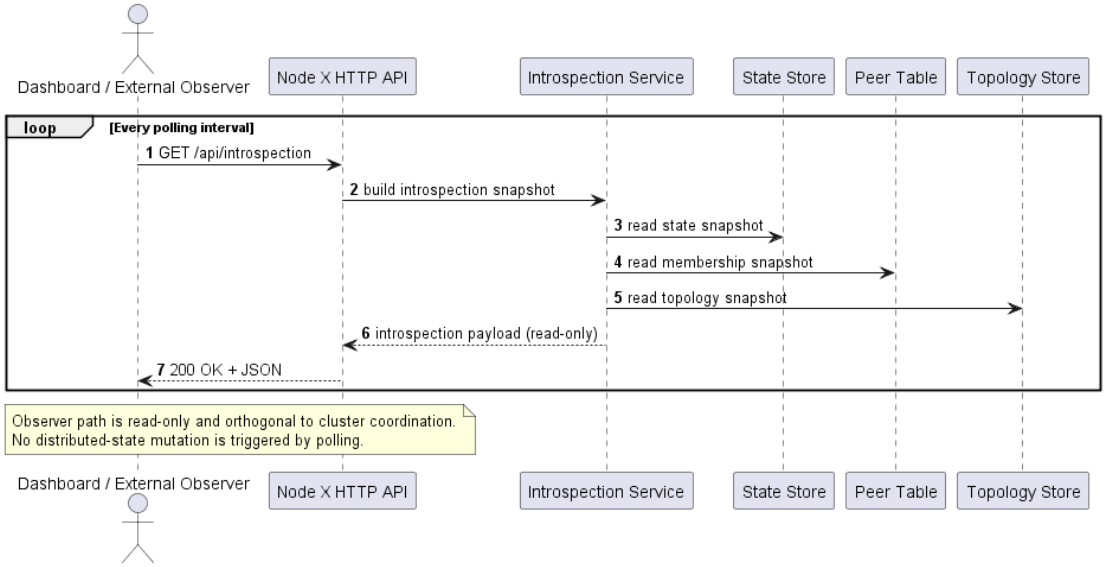

Figure 9: Observer polling over read-only HTTP introspection endpoints.

This design supports replication, decentralized membership/liveness, recovery, and read-only monitoring because it combines low-coupling peer communication with an observation interface that stays orthogonal to the replicated core.

3.5 Behaviour

The behavior of the main components can be summarized as follows.

Sensor sources are active components: they periodically generate local readings and inject them into the node. They are conceptually producers and do not need global knowledge.

The local state manager is stateful and authoritative for the node's replica. More precisely, it is authoritative for local Last-Write-Wins winners and for the node's replication-delta view. It receives local and remote updates, applies deterministic merge rules, stores the current winners, and exposes snapshots to both replication and observability components. This component is central to ingestion, convergence, and deterministic conflict resolution.

The membership manager is also stateful. It stores known peers, updates direct or indirect liveness evidence, and maintains the node's local view of cluster membership.

The failure detector behaves as a local evaluator rather than as a replicated authority. It interprets heartbeat timing and produces suspicion levels for peers observed directly by the node. Those local judgements are then translated into membership information that can be disseminated.

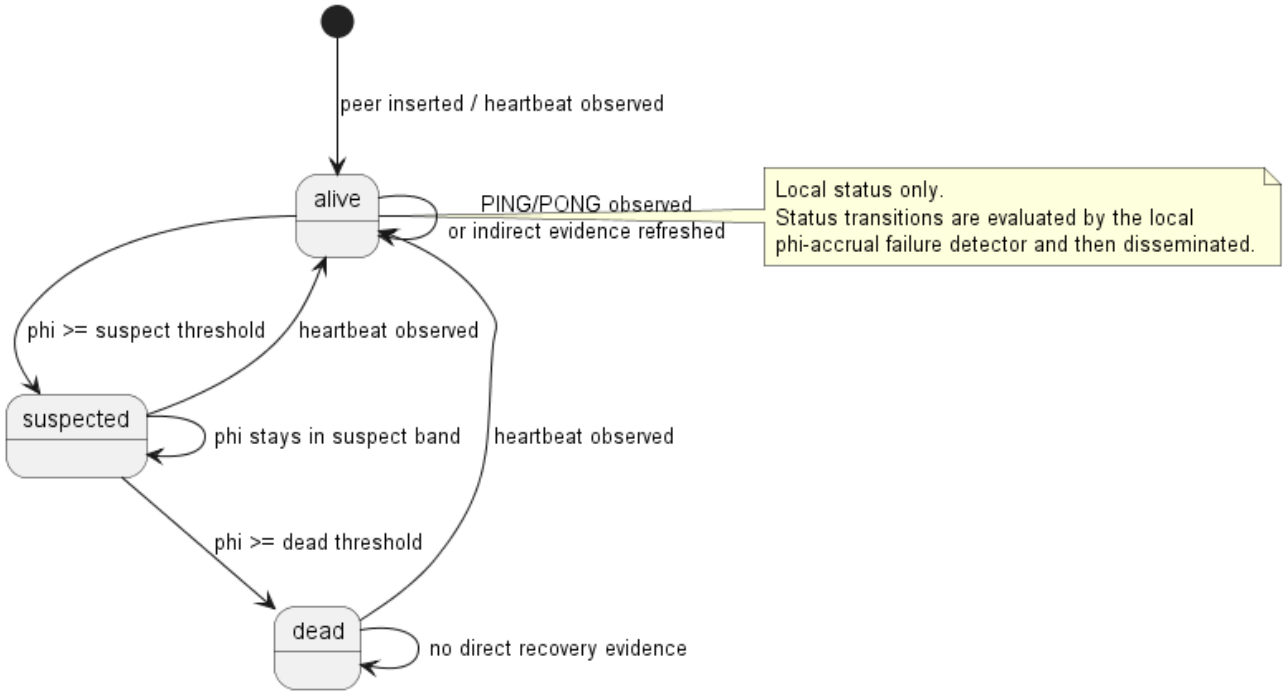

Figure 10: Behavioural view of membership state transitions driven by failure-detection signals.

The heartbeat and gossip runtime is periodic. At each round, it evaluates liveness, sends heartbeat probes, and disseminates updated membership knowledge. Its purpose is not to guarantee exact membership synchronization at every instant, but to make peer-status knowledge progressively converge.

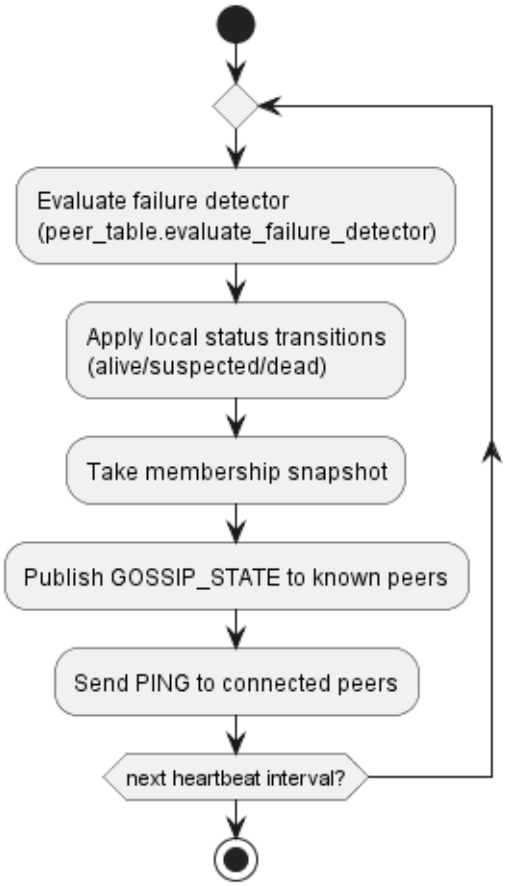

Figure 11: Behavioural activity flow of heartbeat and gossip runtime rounds.

The replication runtime is periodic as well. It pushes recent sensor-state updates to selected peers and pulls missing updates from others. In particular, the publisher side targets peers currently considered `alive`, while heartbeat and membership gossip activities may still attempt communication toward peers that are not currently classified as `alive`. When incremental recovery is not sufficient, it can trigger a broader synchronization phase. This behavior is essential for convergence, recovery, and partition-aware operation.

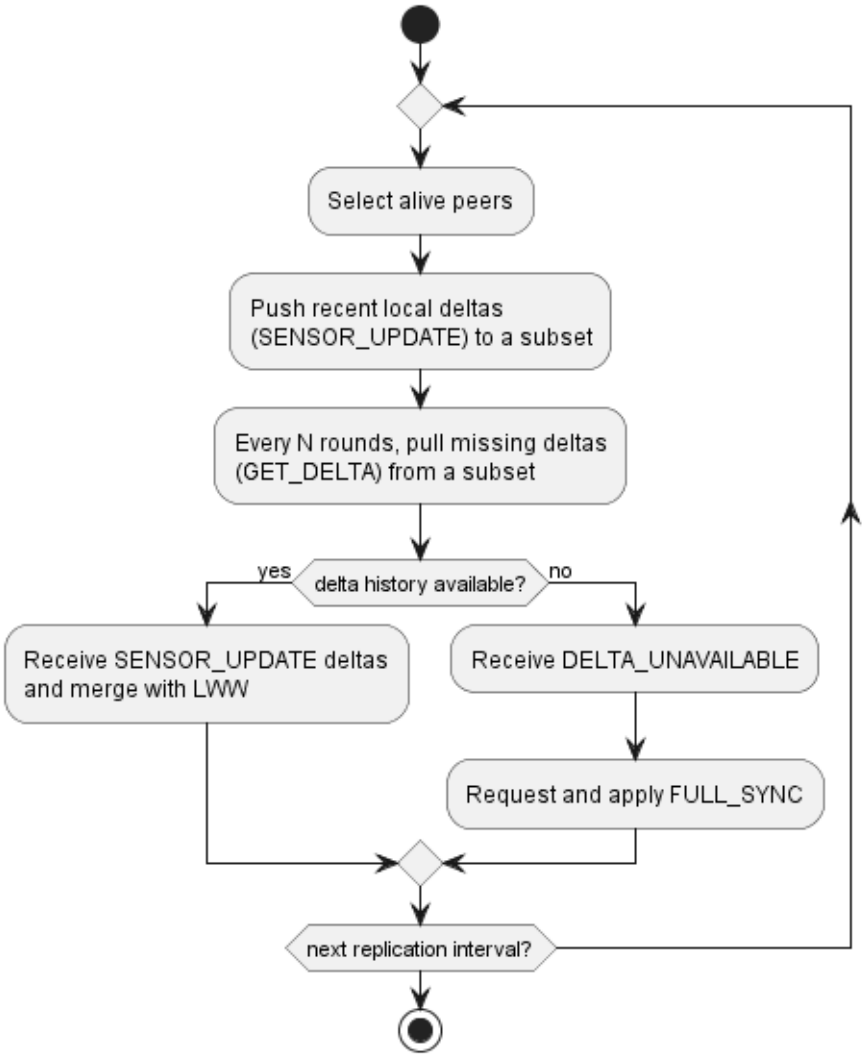

Figure 12: Behavioural activity flow of periodic replication and recovery decisions.

The observability subsystem is read-only. It collects snapshots from state, membership, topology, and event/metric providers, then publishes a coherent observation view for users and tools.

State updates are therefore concentrated in a small set of stateful components:

- local sensor-state storage;
- local membership storage;
- local topology and observability stores.

Other components mainly trigger transitions or move data between these stores. This separation keeps write paths explicit and supports maintainability.

3.6 Data and Consistency Issues

The system keeps only node-local runtime state:

- replicated sensor winners and replication delta metadata;
- local membership/liveness view;
- local topology view;
- recent introspection events and metrics.

No external database is required in the current design. State is in-memory and process-lifetime, with bounded retention for replication history and observability metadata.

There is no persistent dataset in the current implementation. If persistence is introduced in future iterations, sensor winners naturally map to a key-value model, while historical event/metric streams are better represented as append-oriented records.

Because no database is part of the architecture, no component performs database queries in the current design.

Consistency is eventual and data-type specific:

- sensor state uses deterministic LWW merge semantics;
- membership and topology converge through periodic exchange and gossip;
- observability is read-only and snapshot-based by design.

Concurrent access is expected and confined to node-local stores protected by implementation-level synchronization. No distributed transaction spans sensor, membership, topology, and observability data.

3.7 Fault-Tolerance

Fault tolerance is achieved through four complementary mechanisms:

- decentralized state replication across peers;
- heartbeat-based local liveness evaluation with phi-accrual suspicion levels;
- membership gossip for peer-status dissemination;
- anti-entropy recovery via incremental delta sync and full-sync fallback.

Failures are treated as local and recoverable. Under outages or temporary disconnections, surviving nodes continue local operation, membership status may degrade, and synchronization reconciles missed state when connectivity returns.

3.8 Availability

Availability is achieved structurally through node autonomy, replicated state, and per-node read access.

There is no explicit caching layer: each node serves reads from its local replicated state, which provides read locality without a dedicated cache subsystem.

No load balancer is required at the current scale because clients can query any reachable node directly.

Each running node remains a locally available point that can:

- ingest its own local sensor readings;
- maintain its local replica;
- expose read-only snapshots to operators and tools.

There is no mandatory central ingress for observation; clients may query any reachable node.

During a network partition:

- nodes continue operation within their connected component;
- replicas may diverge temporarily;
- remote peers can be marked as suspected/dead.

After healing, incremental dissemination and recovery progressively re-align replicas.

3.9 Security

Security is intentionally limited in the current design and assumes controlled/trusted environments.

Authentication and fine-grained authorization are not core architectural dependencies at this stage. Nodes exchange protocol traffic, while external clients are limited to read-only HTTP observation.

No dedicated cryptographic scheme (e.g., token verification, payload encryption, or mTLS channel protection) is enforced in the current baseline design.

The design remains modular enough to add stronger security controls at clear boundaries:

- node-to-node channel authentication/protection;
- access control on observation endpoints;
- deployment-level isolation and reverse-proxy policies.

4 Implementation

This chapter reports the main technology-dependent choices used to realize the design in section 3.

4.1 Protocols and Data Representation

The implementation uses two communication planes with different responsibilities.

Node-to-node communication. Peer communication is implemented over **TCP**, with a custom application protocol and JSON payloads. TCP is used for:

- bootstrap and peer discovery (`JOIN_REQUEST`, `PEER_LIST`);
- heartbeat liveness traffic (`PING`, `PONG`);
- membership/topology/state view convergence (`GOSSIP_STATE`, distinct from heartbeat and discovery flows);
- replicated sensor updates (`SENSOR_UPDATE`);
- incremental and full catch-up (`GET_DELTA`, `DELTA_UNAVAILABLE`, `FULL_SYNC_REQUEST`, `FULL_SYNC_RESPONSE`).

Replication is implemented as a push-pull runtime: nodes periodically push local deltas and periodically request missing deltas from sampled alive peers, with fallback to full sync when incremental history is unavailable.

Observer communication. External users/tools interact through a **HTTP API** implemented with `ThreadingHTTPServer`. The HTTP surface is intentionally observation-oriented: it exposes read endpoints (e.g., state, updates, membership, topology, introspection) and also serves static dashboard assets, while remaining separate from peer coordination logic.

In-transit representation. Protocol messages and HTTP responses are represented as **JSON** (UTF-8), chosen for inspectability and interoperability with browser/Python tooling.

4.2 State Management and Storage Choices

The implementation uses **in-memory state only**; no external database is used. Each node maintains:

- current winning sensor records;
- bounded ordered replication deltas and per-origin sequence tracking;
- peer membership/liveness state;
- topology state;
- introspection snapshots (events/metrics).

Sensor convergence is implemented in application logic via deterministic LWW merge semantics on `(timestamp, origin)`. Membership and liveness are maintained as local views updated by protocol handlers and periodic runtime components.

Membership and topology runtime model. Membership is maintained per node in a local `PeerTable`, updated by discovery traffic (`JOIN_REQUEST`, `PEER_LIST`), heartbeat evidence (`PING`/`PONG`), and gossip exchanges. Peer status is not binary up/down: it is evaluated as `alive`/`suspect`/`dead` from time-based evidence and configurable phi thresholds (e.g., `phi_threshold_suspect`, `phi_threshold_dead`).

Membership updates use deterministic timestamp-based merge logic for liveness/status information, so node-local views still converge when updates are received in different orders. `GOSSIP_STATE` carries not only peer presence but also metadata useful for convergence of cluster-view, observed topology state, and node-reported state summaries used by the introspection layer.

Topology is represented as explicit runtime state in a dedicated in-memory component (`TopologyStateStore`), updated by gossip/runtime flows and exposed as a read-only snapshot. Link-selection behavior is policy-driven (currently `full_mesh`), configured through `TOPOLOGY_POLICY`, so topology behavior can evolve without changing the peer protocol.

For observability, membership and topology snapshots are exposed through HTTP endpoints (e.g., `/api/membership`, `/api/topology`). Unlike sensor-state LWW records, membership/topology follow dedicated liveness/operational merge rules and are not treated as ordinary sensor-value entities.

4.3 Authentication and Authorization

No explicit authentication/authorization is implemented in the current version (scope choice for controlled lab/development setups):

- peer channels are not authenticated;
- HTTP endpoints do not require tokens/sessions/login;
- no RBAC/ABAC layer is implemented.

There is currently no TLS/mTLS protection on transport channels. Therefore, reachability to an exposed node interface implies access to that interface. Security hardening is future work.

4.4 Technological details

Core technologies used:

- **Python** runtime and standard library (`socket`, `threading`, `ThreadingHTTPServer`, `json`, `logging`);
- **thread-based concurrency** (server threads, worker pools, periodic background threads, queue-based local sensor ingestion);

- **environment-based configuration** via `.env` and `python-dotenv`;
- **Docker/Docker Compose** for reproducible multi-node execution;
- **pytest** and **pyright** for testing and static analysis.

Key implementation mechanisms:

- explicit TCP application framing with server-side frame-size validation;
- inbound backpressure: the node accepts only a bounded number of connections/workers and rejects excess load;
- resilient outbound channels: per-peer message queues with retry/backoff and automatic reconnect;
- push-pull replication tuning via configurable probabilistic fanout/cadence parameters (push ratio, pull ratio, pull cadence);
- bounded incremental delta log (`replication_delta_maxlen`) to control memory usage and trigger full-sync fallback when history is insufficient;
- per-origin cursor tracking (`latest_seq_by_origin`) to resume incremental recovery from correct sequence points;
- configurable runtime fault injection (delay/jitter/loss) for robustness testing without external fault-injection tooling.

Overall, the stack is intentionally lightweight and explicit, prioritizing observability, reproducibility, and deterministic behavior over framework-heavy abstractions.

In discursive terms, these implementation choices directly realize the requirements discussed in section 2, especially the sections on distributed behavior, resilience under failures, and operational observability.

5 Validation

Validation follows a layered approach: unit tests for local correctness, integration tests for multi-node behavior, Docker-based tests for production-like conditions, and manual acceptance checks for observability quality.

5.1 Automatic Testing

Unit and modular tests. Unit tests are implemented with `pytest` and validate component-level responsibilities:

- protocol parsing/validation/dispatch (`tests/protocol/*`);
- networking robustness and limits (`tests/networking/*`);
- LWW state merge and delta handling (`tests/state/*`);
- membership and phi-accrual failure detection (`tests/membership/*`, `tests/fd/*`);
- topology policies and in-memory topology state (`tests/topology/*`);
- sensor providers (`tests/sensors/*`);
- read-only observation endpoints (`tests/webapi/*`, `tests/introspection/*`).

Rationale: prevent local regressions and enforce core invariants independently of topology.

Harness (in-process integration). In-process integration tests exercise interaction among real nodes coordinated by local harnesses:

- harness component: `tests/integration/cluster_harness.py`;
- harness validation: `tests/integration/test_cluster_harness.py`;
- deterministic convergence scenario: `tests/integration/test_deterministic_state_convergence.py`;
- finite deterministic sensor flow scenario: `tests/integration/test_finite_test_sensor.py`.

These tests target properties that unit tests alone cannot prove: bootstrap behavior, propagation ordering effects, and eventual convergence.

Docker (production-like integration). Docker-based integration tests run in containerized Compose topologies:

- harness component: `tests/integration/docker_cluster_harness.py`;
- Docker availability guards: `tests/integration/docker_requirements.py`;

- `tests/integration/test_docker_deterministic_state_convergence.py`: verifies that, with deterministic workload in a containerized multi-node cluster, all nodes converge to the same final state view.
- `tests/integration/test_docker_partition_reconciliation.py`: verifies that after a temporary network partition, nodes reconcile missing updates (delta/full-sync fallback) and converge again once connectivity is restored.
- `tests/integration/test_docker_crash_restart_recovery.py`: verifies that a restarted node re-joins the cluster, recovers current state from peers, and reaches convergence again without manual intervention.

Rationale: reuse the same deployment style used in demonstrations, reducing environment drift.

The Docker integration scenarios use dedicated Compose configurations, in particular `docker/docker-compose-integration-tests.yml` and `docker/docker-compose-integration-determi yml`.

Execution automation. Automated execution is provided by CI workflows (`.github/workflows/unit-tests.yml`, `.github/workflows/integration-tests.yml`). In the current CI configuration, `.github/workflows/unit-tests.yml` runs unit/modular suites, while `.github/workflows/integration-tests.yml` runs the deterministic in-process integration check (`tests/integration/test_deterministic_state_convergence.py`).

How to run the tests. Main execution commands:

- unit/modular tests:

```
python -m pytest -v -m "not integration"
```

- targeted multi-node convergence:

```
python -m pytest tests/integration/test_deterministic_state_convergence.py -q -s
```

- Docker integration validation (examples):

```
python -m pytest tests/integration/test_docker_deterministic_state_convergence.py -c
python -m pytest tests/integration/test_docker_partition_reconciliation.py -q -s
python -m pytest tests/integration/test_docker_crash_restart_recovery.py -q -s
```

Coverage and rationale. The three-layer strategy (unit, harness, Docker) covers the central project goals:

- protocol correctness and validation;
- deterministic replicated-state convergence;
- membership/liveness dissemination;

- crash, restart, and partition tolerance;
- read-only API and introspection correctness.

In summary, validation provides layered coverage (unit + integration) across protocol, networking, state convergence, membership/topology, and fault resilience. Requirement linkage is explicit in test modules and covers the distributed-behavior, resilience, and observability targets described in section 2.

Out of scope for this validation: formal verification, Byzantine tolerance, security hardening guarantees, and durable recovery after total cluster loss.

5.2 Acceptance test

Manual acceptance tests complement automation and focus on aspects that are difficult to validate through assertions alone:

- readability and coherence of observation outputs (`/api/state`, `/api/membership`, `/api/introspection`);
- dashboard consistency during runtime evolution (`/ui`);
- perceivable behavior during fault/recovery scenarios (including controlled stop/restart via `manual_tests/compose_chaos.py`).

Manual testing remains useful because these checks are partly qualitative (human-facing observability and diagnostic clarity), while automated tests remain the primary correctness mechanism.

6 Deployment

Deployment supports both local execution and reproducible multi-node scenarios through Docker Compose.

6.1 Install from scratch

Required software:

- **Python 3.12** (CI-validated) and **pip**;
- **Docker** and **Docker Compose** for multi-node/container runs;
- optional browser for dashboard/API inspection.

```
git clone https://github.com/AlexSantini10/distributed-sensor-hub.git
cd distributed-sensor-hub
pip install -r requirements.txt
```

Expected outcome: dependencies installed and node runtime ready to start.

6.2 Configuration

Runtime setup is environment-based (`.env.example`). Key variables include:

- identity/endpoints: `NODE_ID`, `HOST`, `PORT`, `WEB_API_PORT`;
- cluster bootstrap: `BOOTSTRAP_PEERS`;
- heartbeat/replication tuning: `HEARTBEAT_INTERVAL_MS`, `GOSSIP_*`;
- failure detection: `PHI_THRESHOLD_*`;
- optional sensor/network simulation parameters.

6.3 Run

Single node (example):

```
python node.py
```

Expected outcome: active TCP listener + HTTP read API (e.g. `/api/introspection`).
Multi-node with Docker Compose (examples):

```
docker compose -f docker/docker-compose-6-nodes.yml up --build -d
docker compose -f docker/docker-compose-12-nodes.yml up --build -d
```

Expected outcome: one container per node, each with dedicated identity/ports and cluster bootstrap configuration.
As an alternative to local image build, a prebuilt image can be pulled directly from GHCR:

```
docker pull ghcr.io/alexsantini10/distributed-sensor-hub:latest
```

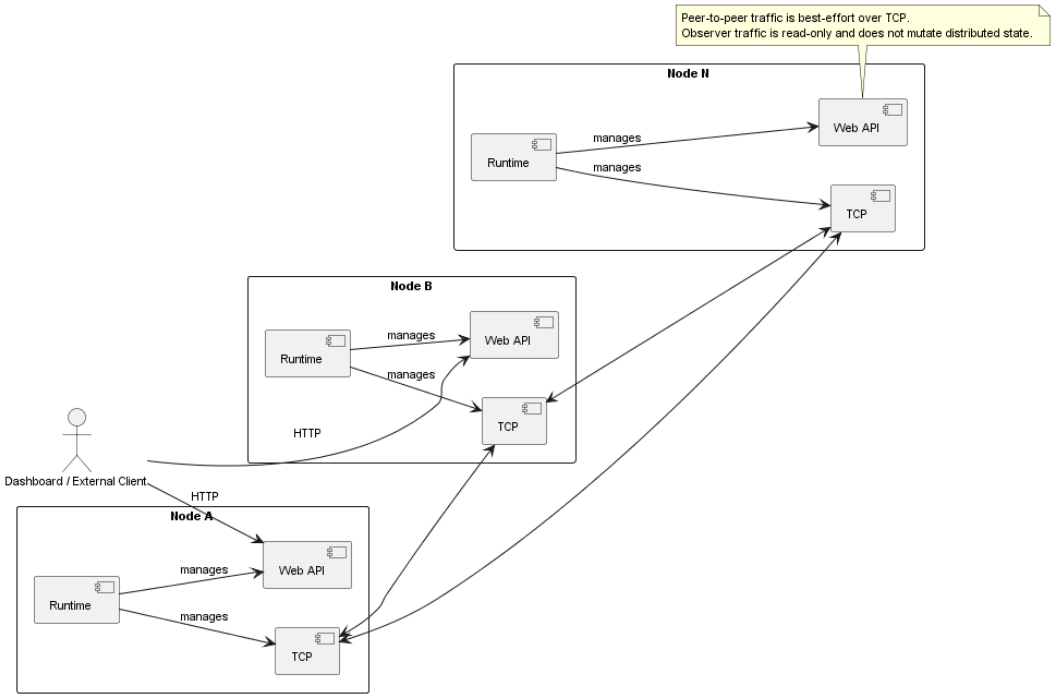

Figure 13: Deployment view of a multi-node cluster with peer TCP channels and read-only observer access.

7 User Guide

This guide describes three practical usage scenarios: two real usage paths and one distributed test/demo path.

7.1 Prerequisites

- Docker Desktop (with Docker Compose);
- for native local execution only: Python 3.12+ and `pip`;
- available TCP ports from `9000+` and HTTP ports from `10000+`, depending on scenario.

7.2 Scenario A (real use): published GHCR image

Goal: quickly start a node from the already published container image.

1. Pull image:

   ```
   docker pull ghcr.io/alexsantini10/distributed-sensor-hub:latest
   ```

2. Start one node:

   ```
   docker run -d --name dsh-node-1 `
   -p 9000:9000 `
   -p 10000:10000 `
   -v "${PWD}:/work" `
   -e NODE_ID=node-1 `
   -e HOST=0.0.0.0 `
   -e PORT=9000 `
   -e BOOTSTRAP_PEERS= `
   -e LOG_LEVEL=INFO `
   -e LOG_FILE=/work/node-1.log `
   -e CLEAR_LOG=true `
   -e WEB_API_PORT=10000 `
   -e SENSORS=1 `
   -e SENSOR_0_TYPE=numeric `
   -e SENSOR_0_NAME=temperature `
   -e SENSOR_0_PERIOD_MS=1000 `
   -e SENSOR_0_MIN=18 `
   -e SENSOR_0_MAX=28 `
   -e SENSOR_0_UNIT=C `
   ghcr.io/alexsantini10/distributed-sensor-hub:latest
   ```

   Start one node (Bash):

```
docker run -d --name dsh-node-1 \
  -p 9000:9000 -p 10000:10000 \
  -e NODE_ID=node-1 \
  -e HOST=0.0.0.0 \
  -e PORT=9000 \
  -e BOOTSTRAP_PEERS= \
  -e LOG_LEVEL=INFO \
  -e LOG_FILE=/app/logs/node-1.log \
  -e CLEAR_LOG=true \
  -e WEB_API_PORT=10000 \
  -e SENSORS=1 \
  -e SENSOR_0_TYPE=numeric \
  -e SENSOR_0_NAME=temperature \
  -e SENSOR_0_PERIOD_MS=1000 \
  -e SENSOR_0_MIN=18 \
  -e SENSOR_0_MAX=28 \
  -e SENSOR_0_UNIT=C \
  ghcr.io/alexsantini10/distributed-sensor-hub:latest
```

Important: run this Docker command from the directory where you want log files to be generated. With `-v "${PWD}:/work"` and `LOG_FILE=/work/node-1.log`, logs are written to the current host directory.

Expected results:

- container state is `Up` (`docker ps`);
- API reachable on `http://localhost:10000/api/state`;
- UI reachable on `http://localhost:10000/ui`.

7.3 Scenario B (real use): repository clone + native node startup

Goal: run one node natively with Python, useful for development/debugging.

1. Clone and install dependencies:

```
git clone https://github.com/AlexSantini10/distributed-sensor-hub.git
cd distributed-sensor-hub
pip install -r requirements.txt
```

2. Start one node (PowerShell):

```
$env:NODE_ID="node-1"
$env:HOST="0.0.0.0"
$env:PORT="9000"
```

```
$env:BOOTSTRAP_PEERS=""
$env:LOG_LEVEL="INFO"
$env:LOG_FILE="logs/node-1.log"
$env:CLEAR_LOG="true"
$env:WEB_API_PORT="10000"
$env:SENSORS="1"
$env:SENSOR_0_TYPE="numeric"
$env:SENSOR_0_NAME="temperature"
$env:SENSOR_0_PERIOD_MS="1000"
$env:SENSOR_0_MIN="18"
$env:SENSOR_0_MAX="28"
$env:SENSOR_0_UNIT="C"
python node.py
```

Start one node (Bash):

```
export NODE_ID="node-1"
export HOST="0.0.0.0"
export PORT="9000"
export BOOTSTRAP_PEERS=""
export LOG_LEVEL="INFO"
export LOG_FILE="logs/node-1.log"
export CLEAR_LOG="true"
export WEB_API_PORT="10000"
export SENSORS="1"
export SENSOR_0_TYPE="numeric"
export SENSOR_0_NAME="temperature"
export SENSOR_0_PERIOD_MS="1000"
export SENSOR_0_MIN="18"
export SENSOR_0_MAX="28"
export SENSOR_0_UNIT="C"
python node.py
```

Expected results:

- Python process runs without critical startup errors;
- HTTP endpoints are available (e.g., `/api/state`, `/api/membership`, `/api/introspection`);
- UI reachable on `http://localhost:10000/ui`.

If startup fails with missing environment variables, use the project `.env.example` as reference and ensure all required keys are defined (at minimum, `LOG_LEVEL` must be set).

7.4 Scenario C (test/demo): repository clone + Docker Compose multi-node

Goal: validate distributed behavior on a multi-node setup.

1. Download and extract the repository (or clone it), then move into the project directory:

```
# Option A (Git)
git clone https://github.com/AlexSantini10/distributed-sensor-hub.git
cd distributed-sensor-hub

# Option B (ZIP)
# Download from GitHub -> Extract ZIP -> open terminal in extracted folder
```

2. From repository root, start demo topology:

```
docker compose -f docker/docker-compose-6-nodes.yml up --build -d
```

3. Optional fault/recovery check:

```
docker compose -f docker/docker-compose-6-nodes.yml stop node-4
docker compose -f docker/docker-compose-6-nodes.yml start node-4
```

4. Stop environment:

```
docker compose -f docker/docker-compose-6-nodes.yml down
```

Expected results:

- all nodes start and become mutually visible in membership/topology views;
- sensor-state replication is visible across nodes through gossip push/pull;
- convergence is visible from multiple UIs (e.g., `http://localhost:10000/ui`, `http://localhost:1000`
- after node restart, missing updates are recovered and convergence is restored.

7.5 Screenshots (report evidence)

Capture screenshots from a real run of the 12-node topology.

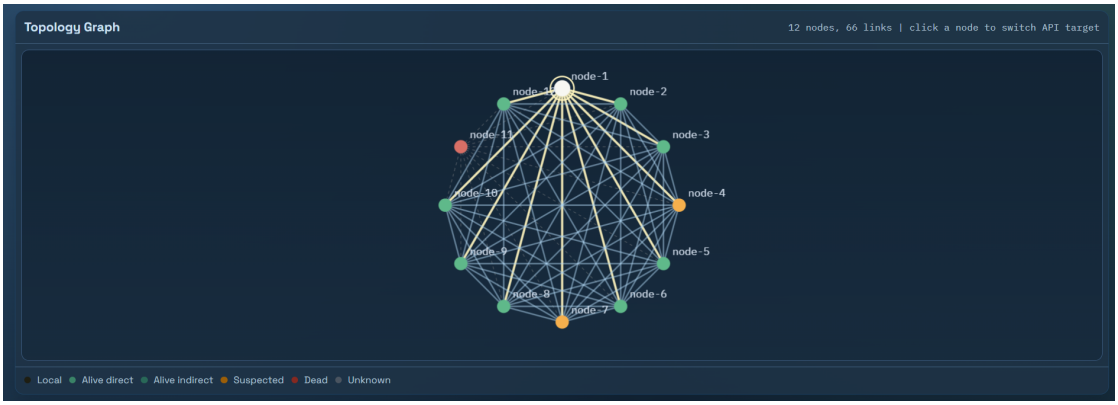

Figure 14: Topology and replication metrics after 12-node convergence.

Selected Node Inspector

node-1 | status: local | direct observed by local node: yes | phi(local): n/a

Peer	Relation	Status	Phi	Evidence
node-2	direct	alive_direct	0.171	direct_heartbeat
node-3	direct	alive_direct	0.093	direct_heartbeat
node-4	direct	suspected	2.649	gossip_status
node-5	direct	alive_direct	0.298	direct_heartbeat
node-6	direct	alive_direct	0.300	gossip_status
node-7	direct	suspected	2.422	direct_heartbeat
node-8	direct	alive_direct	0.000	direct_heartbeat
node-9	direct	alive_direct	0.201	direct_heartbeat
node-10	direct	alive_direct	0.000	direct_heartbeat
node-11	direct	dead	13.934	gossip_status
node-12	direct	alive_direct	0.235	direct_heartbeat

Figure 15: Membership/liveness view (`alive`, `suspected`, `dead`).

Figure 16: Replicated sensor state with source and timestamp.

8 Self-evaluation

This project was developed individually by Alex Santini. This section therefore reflects my full personal contribution.

8.1 Alex Santini

Role in the project. As sole contributor, I covered the full lifecycle: requirements, design, implementation, testing, deployment setup, observability, and report writing.

Main contributions. I implemented a decentralized P2P sensor-state system with TCP + JSON messaging, gossip-based dissemination, LWW merge, membership/failure detection, and recovery (delta/full sync). I also prepared Docker-based multi-node execution, read-only HTTP/introspection endpoints, and layered automated validation (unit, in-process integration, Docker scenarios).

Product strengths. Main strengths are architectural clarity (layered node design), explicit decentralized behavior, reproducible deployment, and good observability/testing coverage for convergence and recovery scenarios.

Weaknesses and limitations. The system is still a course-project prototype: no authentication/authorization/TLS, in-memory state only, and limited quantitative evaluation at larger scale.

Group members.

- Alex Santini – `alex.santini2@studio.unibo.it`

9 Future works

The current implementation satisfies the project objectives, but several extensions could improve production-readiness and experimental depth.

9.1 Security and operational hardening

The current deployment model assumes a trusted network and cooperative nodes. A realistic deployment should strengthen identity, transport security, and endpoint control:

- mutual node authentication to prevent unauthorized peers from joining the cluster;
- protection of HTTP endpoints via tokens and/or mTLS, to restrict data exposure;
- explicit authorization boundaries between read-only observability operations and administrative/runtime control operations.

9.2 Persistence and historical data

At present, node state is in-memory, which is sufficient for runtime convergence but weak for restart continuity and post-mortem analysis. Useful next steps include:

- local persistence of sensor state and recent deltas;
- periodic snapshots of membership and topology views;
- startup restore to reduce cold-start convergence and recovery time;
- optional export of time-series/history data for offline analysis and reproducible experiments.

9.3 Richer replication semantics

The current LWW policy is deterministic and simple, but it is still a general-purpose approximation. Future iterations could evaluate richer consistency semantics:

- CRDTs tailored to specific data types;
- merge policies differentiated by sensor category and domain meaning;
- causal metadata (e.g., version vectors) to preserve ordering information;
- adaptive selection between delta synchronization and full synchronization (refining the existing fallback policy) based on runtime conditions.

9.4 Low-degree topology policy and validation

A key next step is the design and implementation of a low-degree topology policy, as an alternative to full-mesh connectivity. After introducing this policy, the system should be validated experimentally to measure its impact on behavior and scalability. This would allow:

- comparing full-mesh and low-degree policies under the same workloads;
- quantifying the trade-off between reduced connection density and convergence speed;
- measuring effects on gossip overhead, sync traffic, and recovery dynamics;
- analyzing failure-detector sensitivity under different latency/loss distributions;
- tuning fan-out, synchronization intervals, and delta-buffer size under sustained load.