Gossip-Based Distributed Monitoring System

Final Report for the Distributed Systems Course [2025/2026]

Thomas Testa
thomas.testa@studio.unibo.it

May 26, 2026

This project presents the design and implementation of a fully decentralized monitoring system based on gossip (epidemic) protocols [3].

The system consists of multiple independent nodes that collaboratively maintain cluster membership and detect failures without relying on any centralized coordinator. Each node periodically exchanges partial state information with a subset of randomly selected peers. Through these probabilistic interactions, membership updates propagate across the system, enabling nodes to converge toward a consistent global view even in the presence of network delays, packet loss, and node crashes. Failure detection is achieved through heartbeat mechanisms combined with timeout-based state transitions (`ALIVE`, `SUSPECT`, `DEAD`),following a design inspired by the SWIM protocol [2]. The system is designed to operate in asynchronous and unreliable environments while ensuring eventual consistency. An experimental extension, called *Gossip Arena*, is introduced to study information dissemination dynamics. Synthetic rumor events are injected into the system to evaluate the impact of different gossip strategies and parameters on dissemination speed and communication overhead. Experimental results show that the system remains robust under moderate network impairments and highlight the trade-offs between convergence speed, communication cost, and accuracy of failure detection.

Disclaimer

During the preparation of this work, the author used AI-based tools to support writing, restructuring, language refinement, and the generation of explanatory diagrams and visual assets included in the report. All experimental results, measurements, validation procedures, execution logs, and performance analyses were produced through direct implementation, deployment and testing activities carried out by the author on the developed distributed system. After using AI-assisted tools, the author carefully reviewed,

validated, and edited the generated material as needed, and takes full responsibility for the content and correctness of the final report.

1 Concept

1.1 Product Description

This project implements a **distributed monitoring service** in the form of a set of cooperating command-line processes (nodes), each running as an independent containerized service. There is no graphical user interface; the system is designed to be used programmatically and observed through structured logs and metrics.

The primary use case is the monitoring of a dynamic cluster of nodes where:

- nodes may join or leave the cluster at any time;
- the network is unreliable (packet loss, delays);
- no centralized coordinator is available or desired.

Users (typically system operators or researchers) interact with the system indirectly by reading logs, configuring environment variables before deployment, and optionally stopping containers to simulate failures. Interaction happens at deployment time and during observation; the system itself runs autonomously once started.

The system follows an epidemic communication model: nodes periodically exchange partial membership information with randomly selected peers using gossip rounds. Through repeated dissemination, cluster state changes (heartbeats, suspicions, failures) progressively propagate across the network until convergence is reached.

No persistent user data needs to be stored. The system's state (membership tables, heartbeat counters, and node statuses) is kept entirely in-memory at each node and is considered ephemeral.

Rather than enforcing strong consistency through centralized coordination, the system adopts an **eventual consistency** model. Due to asynchronous communication and unreliable delivery, nodes may temporarily observe different cluster views. However, repeated gossip exchanges guarantee probabilistic convergence toward a coherent global membership state.

The project is intentionally implemented using low-level networking primitives and decentralized coordination mechanisms in order to study core distributed systems concepts such as epidemic dissemination, failure detection, partial replication, probabilistic convergence, and fault tolerance without relying on external orchestration frameworks.

1.2 Why Distribution Is Needed

Distribution is the core requirement of this system, not merely an implementation detail. Specifically:

- **Fault tolerance**: a centralized monitor would represent a single point of failure. If the monitor crashes, cluster visibility is lost entirely. A decentralized approach ensures that every node independently maintains a partial view of the cluster, so no single failure can compromise the monitoring functionality.
- **Distributed failure detection**: in asynchronous systems, node crashes cannot be directly observed and must instead be inferred indirectly through heartbeat timeouts and probabilistic dissemination. This project explores how decentralized nodes collectively construct a coherent view of failures despite unreliable communication.
- **Scalability**: gossip-based dissemination scales efficiently with the number of nodes [5]. Each node contacts only a small, configurable subset of peers (fanout k) during each gossip round, reducing communication overhead compared to fully connected synchronization approaches.
- **Geographically distributed environments**: the system is designed for scenarios where nodes run across different machines or containers with independent clocks and no shared memory, communicating exclusively through message passing.
- **Resource sharing and replication**: membership information is collaboratively maintained and replicated across the cluster. No node possesses complete knowledge initially; instead, nodes continuously exchange partial views to reconstruct the global system state through gossip propagation.
- **Educational value**: the project serves as a practical exploration of distributed systems concepts such as eventual consistency, decentralized coordination, epidemic communication, failure detection, and probabilistic convergence under unreliable network conditions.

2 Requirements Elicitation and Analysis

2.1 Functional Requirements

- **FR1: Gossip-Based Communication**
  Each node must periodically exchange membership information with a subset of peers using a gossip-based dissemination protocol.
  Acceptance criterion: During each gossip round, nodes successfully exchange heartbeat and membership information, observable through structured logs and metrics.
- **FR2: Distributed Membership Management**
  Each node must maintain a local membership table containing known peers, heartbeat counters, incarnation numbers, and node states (`ALIVE`, `SUSPECT`, `DEAD`).
  Acceptance criterion: After a sufficient convergence period, all reachable nodes report equivalent membership views.

- **FR3: Failure Detection**
  The system must detect crash failures through heartbeat monitoring and timeout-based state transitions.
  Acceptance criterion: After intentionally stopping a node, the remaining nodes eventually transition that node to the `DEAD` state within a measurable detection latency.

- **FR4: Distributed State Dissemination**
  Membership updates and failure information must propagate across the cluster through epidemic dissemination.
  Acceptance criterion: A state update generated by one node eventually reaches all reachable nodes within a bounded number of gossip rounds.

- **FR5: Dynamic Membership**
  Nodes must be able to dynamically join the cluster and become visible to existing peers without requiring centralized reconfiguration.
  Acceptance criterion: Newly started nodes appear in the membership tables of active peers after a finite convergence interval.

- **FR6: Fault Injection Support**
  The system must support controlled simulation of unreliable network conditions, including packet loss and message delay.
  Acceptance criterion: Configuring `PACKET_LOSS_RATE` and `MESSAGE_DELAY_MS` produces observable changes in dissemination latency, convergence speed, and communication reliability.

- **FR7: Gossip Arena Experiments**
  The system must support synthetic rumor dissemination experiments for evaluating epidemic propagation dynamics.
  Acceptance criterion: Injected rumors propagate across the cluster and their dissemination metrics (coverage, latency, hop count, propagation rounds) can be measured experimentally.

2.2 Non-Functional Requirements

- **Scalability**: the system should scale to tens of nodes without centralized coordination, while maintaining bounded per-node communication costs.

- **Fault Tolerance**: the monitoring service must continue operating despite crash failures and unreliable network conditions.

- **Eventual Consistency**: because the system operates in an asynchronous environment with probabilistic dissemination, nodes may temporarily observe divergent membership views; however, the system must converge toward a coherent global state over time.

- **Availability**: nodes must continue exchanging membership information and performing failure detection even when portions of the network become temporarily unreachable.
- **Performance Efficiency**: the system must balance convergence speed, detection latency, and communication overhead through configurable dissemination parameters.
- **Configurability**: gossip fanout, heartbeat interval, suspicion timeout, dissemination frequency, and fault injection settings must be adjustable through environment variables.
- **Observability**: the internal behavior of the distributed system must be externally observable through structured logs and metrics to facilitate debugging, experimentation, and performance analysis.

2.3 Implementation Requirements

- The system is implemented in Python 3 using asynchronous programming through the `asyncio` framework.
- Inter-node communication is implemented using UDP sockets in order to model unreliable message-oriented communication.
- Docker and Docker Compose are used to simulate distributed multi-node execution environments.
- JSON is used for lightweight and human-readable message serialization.
- The implementation intentionally avoids external distributed coordination frameworks in order to directly study low-level distributed systems mechanisms.

2.4 Relevant Distributed System Features

- **Transparency**: distribution is intentionally *not* hidden. Nodes explicitly operate with partial and potentially outdated knowledge of the cluster. This design choice emphasizes the realities of distributed computation.
- **Fault Tolerance and Dependability**: the system tolerates crash failures and unreliable communication through decentralized failure detection and replicated membership information.
- **Scalability**: gossip dissemination bounds the communication overhead to approximately $O(k)$ messages per node per round, where k is the configurable fanout parameter.
- **Consistency and Replication**: membership information is replicated across nodes using epidemic dissemination. The system favors availability and eventual consistency over strong synchronization guarantees.

- **Asynchrony**: nodes operate independently without synchronized clocks or shared memory, communicating exclusively through asynchronous message passing.
- **Security and Trust**: security is intentionally out of scope. The system assumes a cooperative and non-malicious environment; therefore, authentication and encryption mechanisms are not implemented.
- **Resource Sharing**: nodes collaboratively maintain the cluster membership state by continuously exchanging partial views of the system.
- **Openness and Interoperability**: the use of standard technologies (UDP, JSON, Docker, Python) allows the system to run across heterogeneous platforms.
- **Maintainability and Evolvability**: networking, dissemination, membership management, and failure detection are implemented as modular and independent components, simplifying future extensions.
- **Performance and Efficiency**: the primary performance metrics are convergence time, dissemination coverage, detection latency, and communication overhead, all analyzed experimentally through the Gossip Arena.

3 Design

3.1 Architecture

The system adopts a fully decentralized **peer-to-peer** architectural style based on gossip (epidemic) protocols. The design is strongly inspired by the SWIM (*Scalable Weakly-consistent Infection-style Process Group Membership*) protocol, particularly regarding decentralized membership dissemination, failure suspicion, and indirect propagation of node state transitions. There is no central coordinator, master node, or shared database. Each node is architecturally identical and operates independently, maintaining a partial view of the cluster. This choice is motivated by:

- **Elimination of single points of failure**: any node can fail without affecting the ability of the remaining nodes to monitor the cluster.
- **Scalability**: decentralized communication does not create bottlenecks or centralized coordination overhead.
- **Homogeneity**: all nodes execute the same logic and participate equally in dissemination and failure detection, simplifying deployment and maintenance.

In the current implementation, nodes support both **push** and **push-pull** epidemic dissemination strategies. In push mode, nodes periodically transmit local membership updates to randomly selected peers. In push-pull mode, contacted peers additionally respond with their own membership view, enabling bidirectional anti-entropy synchronization and improving convergence speed under unreliable network conditions.

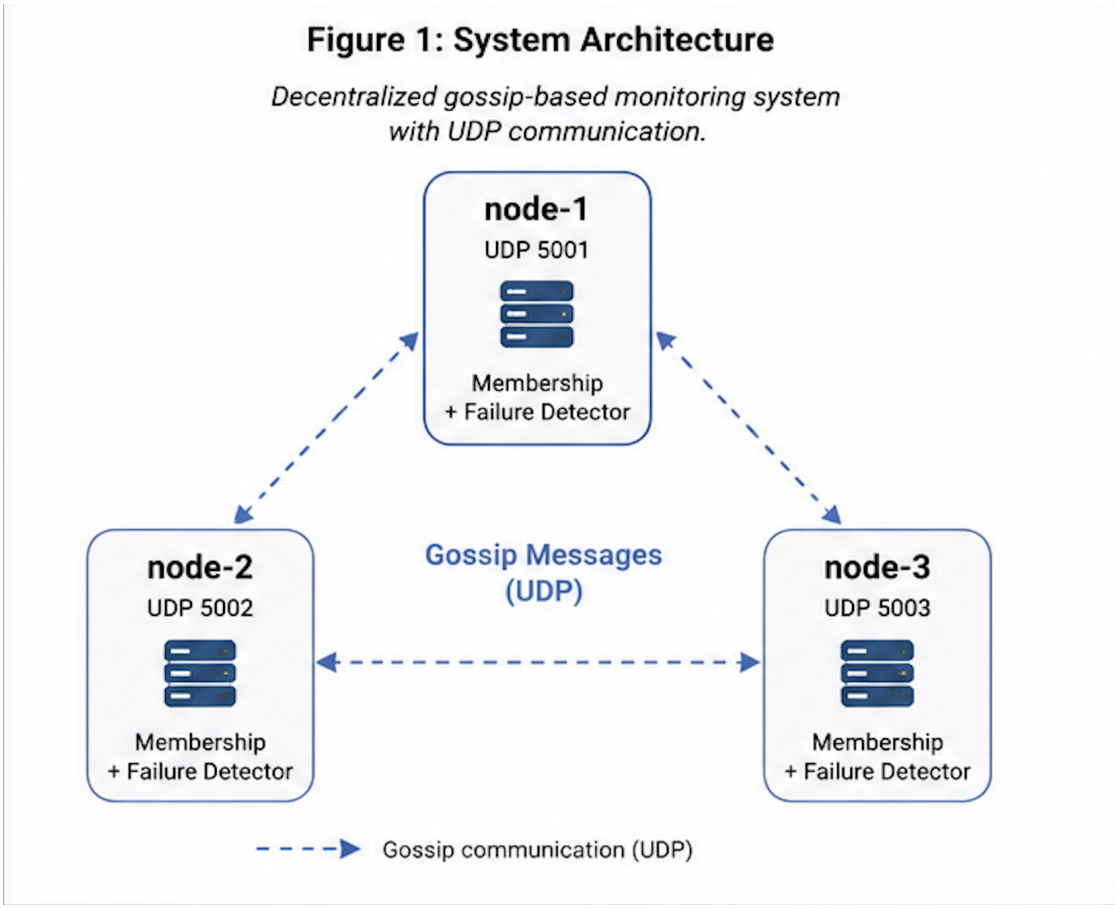

Figure 1: Decentralized gossip-based monitoring architecture.

3.2 Infrastructure

The system is composed exclusively of **homogeneous nodes**. Each node includes four logical modules:

1. **Network Layer**: handles UDP socket creation, message sending, and message reception.
2. **Gossip Engine**: periodically selects random peers (fanout k) and drives epidemic dissemination rounds.
3. **Membership Manager**: maintains the local membership table and applies merge policies for received updates.
4. **Failure Detector**: monitors heartbeat freshness and triggers timeout-based state transitions.

Nodes are deployed as Docker containers in a shared virtual network managed by Docker Compose. Peer discovery relies on a predefined seed list of known nodes passed through environment variables at startup. Each node is identified by a unique `NODE_ID` and is reachable through its container hostname and port. UDP was intentionally chosen over TCP because gossip dissemination prioritizes low overhead and probabilistic redundancy rather than reliable point-to-point delivery. Packet loss is tolerated through repeated dissemination rounds, making reliability emerge statistically rather than through retransmission.

3.3 Modelling

Domain entities:

- **Node**: an independent process participating in the system, identified by `node_id`, `address`, and `port`.
- **Membership Entry**: a local record describing a peer, containing:`node_id`, `heartbeat` counter, `incarnation` number, `status` (`ALIVE`, `SUSPECT`, `DEAD`) and `updated_at` timestamp.
- **Gossip Message**: a serialized JSON payload containing a subset of the sender's membership table.
- **Rumor Event**: a synthetic event injected during Gossip Arena experiments, identified by a unique `rumor_id` and originating node.

All entities are managed locally by each node. No centralized storage exists.The membership table represents the entirety of the distributed system state and is replicated implicitly through repeated gossip dissemination.

Domain events:

- Gossip Round Triggered
- Heartbeat Incremented
- Membership Update Received and Merged
- Node Suspected (`ALIVE` $\rightarrow$ `SUSPECT`)
- Node Declared Dead (`SUSPECT` $\rightarrow$ `DEAD`)
- Rumor Generated / Received / Disseminated

3.4 Interaction

Nodes communicate using configurable **push** and **push-pull gossip** protocols. During each gossip round, nodes periodically exchange membership information with a subset of randomly selected peers.

1. The node increments its local heartbeat counter.
2. A subset of peers is selected randomly from the local membership table according to the configured fanout parameter k.
3. The node serializes its current membership information into a JSON message.
4. The serialized membership state is disseminated to the selected peers through UDP communication.
5. Upon receiving a gossip message, the receiver merges the incoming entries into its own membership table according to the merge policy.

Merge policy:
For each received membership entry, the local entry is updated if at least one of the following conditions holds:

- the received heartbeat counter is strictly greater than the local one;
- the received incarnation number is greater than the local incarnation number, allowing a node to refute obsolete suspicion states;
- the received status is `DEAD`, which is considered a terminal and monotonic state.

Formally, given a local entry L and a received entry R, R supersedes L if:

$$(R.hb > L.hb) \ \vee \ (R.inc > L.inc) \ \vee \ (R.status = \texttt{DEAD}) \tag{1}$$

From a communication complexity perspective, each node sends at most k gossip messages per dissemination round, where k is the configurable fanout parameter. Consequently, the per-node communication cost remains bounded independently from total cluster size, while dissemination latency decreases as fanout increases. This introduces a tradeoff between convergence speed and bandwidth usage.

3.5 Behaviour

Each membership entry follows the finite state machine shown in fig. 2. A node is initially considered `ALIVE`. If no heartbeat is received within the suspicion timeout, the node is marked as `SUSPECT`. If the suspicion is not refuted before the dead timeout expires, the node is finally marked as `DEAD`.

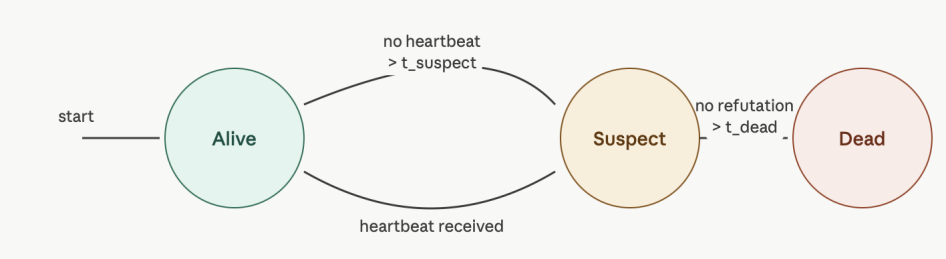

Figure 2: Finite state machine governing node membership states.

Each node continuously executes two asynchronous activities: a gossip loop, which disseminates membership information, and a timeout checker, which updates membership states according to heartbeat freshness.

Listing 1: Simplified gossip and failure detection logic

```
async def gossip_loop():
    while True:
        increment_local_heartbeat()

        peers = select_random_peers(
            fanout=FANOUT
        )

        for peer in peers:
            await send_udp(
                peer,
                serialize(membership)
            )

        await asyncio.sleep(
            GOSSIP_INTERVAL
        )

def check_timeouts():
    now = time.time()

    for entry in membership.values():
        age = now - entry.updated_at

        if age > SUSPECT_TIMEOUT \
           and entry.status == ALIVE:
            entry.status = SUSPECT

        elif age > DEAD_TIMEOUT \
             and entry.status == SUSPECT:
            entry.status = DEAD
```

3.6 Data and Consistency Issues

The system does not rely on persistent storage. All state is maintained in-memory and replicated probabilistically through epidemic dissemination. Consistency is intentionally **eventual**: due to asynchronous communication, packet loss, and independent gossip schedules, nodes may temporarily observe different membership views. For example, one node may already consider a peer `DEAD` while another still marks it as `ALIVE`.Repeated gossip dissemination progressively reduces these inconsistencies and drives the cluster toward probabilistic convergence. No strong consistency guarantees are enforced. The design intentionally favors availability and partition tolerance over synchronization strictness, in line with the CAP theorem [1].

3.7 Fault-Tolerance

Fault tolerance is achieved through decentralization, redundancy, and repeated information dissemination.

- Heartbeat freshness is used to infer node failures indirectly.
- Timeout thresholds trigger monotonic state transitions: `ALIVE` $\rightarrow$ `SUSPECT` $\rightarrow$ `DEAD`.
- Gossip dissemination propagates failure information across all reachable nodes.
- Packet loss and temporary delays are compensated by repeated gossip rounds. The probability that an update is never disseminated decreases exponentially over time.
- Incarnation numbers allow nodes to refute outdated suspicion states and recover from transient communication issues.

The system tolerates crash failures, message loss, and network delays. Byzantine (malicious) failures are explicitly outside the project scope.

3.8 Availability

The system is highly available because no centralized component is required for monitoring or membership maintenance.

In the presence of a **network partition**:

- Nodes within each partition continue operating autonomously.
- Membership views diverge while connectivity is unavailable.
- Once communication is restored, gossip dissemination progressively reconciles the divergent views until convergence is reached.

This behavior aligns with an AP(*Availability + Partition Tolerance*) interpretation of the CAP theorem, accepting eventual rather than strong consistency.

3.9 Security

Security is intentionally outside the scope of this project. The system assumes a cooperative and non-malicious environment. Therefore, no authentication, authorization, encryption, or Byzantine fault tolerance mechanisms are implemented.

4 Implementation

- **Network Protocol**: UDP sockets are used for all inter-node communication. UDP was chosen due to its low overhead and natural fit for gossip protocols, where occasional message loss is tolerable and handled at the application level through redundant dissemination.
- **Data Representation**: messages are serialized using JSON. This format was selected for readability, ease of debugging, and simplicity of implementation. Each gossip message is a JSON array of membership entries.
- **Persistence**: none. The system stores no data on disk. All state is ephemeral and in-memory.
- **Authentication / Authorization**: none. The system targets a trusted experimental environment.

4.1 Technological Details

- **Python 3.11**: main implementation language.
- **asyncio**: provides the event loop and concurrency model. All I/O (UDP receive, periodic timers) is non-blocking and handled through coroutines.
- **UDP sockets** via `asyncio.DatagramProtocol`: lightweight communication between nodes.
- **FastAPI**: exposes lightweight REST endpoints for querying membership state, runtime metrics, rumor dissemination data, and injecting synthetic Gossip Arena events during experimentation and debugging.
- **Docker**: each node is containerized with its own isolated network stack.
- **Docker Compose**: defines the multi-node topology, assigning unique ports and environment variables to each container service.

Configuration parameters are passed as environment variables:

Variable	Description	Default
NODE_ID	Unique node identifier	required
BIND_PORT	UDP port to listen on	required
PEERS	Comma-separated list of peer addresses	required
GOSSIP_INTERVAL	Seconds between gossip rounds	1.0
FANOUT	Number of peers contacted per round	2
SUSPECT_TIMEOUT	Seconds before ALIVE → SUSPECT	5.0
DEAD_TIMEOUT	Seconds before SUSPECT → DEAD	10.0
PACKET_LOSS_RATE	Probability [0,1] of dropping a message	0.0
MESSAGE_DELAY_MS	Artificial delay in milliseconds	0

Module structure:

```
src/
    main.py                              # Application entry point
    node/node.py                         # Core node logic, gossip
        execution and rumor dissemination
    node/membership.py                   # Membership table and
       merge policies
    network/udp.py                       # UDP communication and
       fault injection
    protocol/failure_detector.py         # Timeout-based failure
       detection
    dashboard/api.py                     # FastAPI REST endpoints
       and rumor injection
```

5 Validation

5.1 Experimental Setup

The system was experimentally evaluated using a local multi-node deployment on macOS. Each node was executed as an independent Python process communicating through UDP sockets over the loopback interface. Experiments were conducted using clusters composed of three nodes. The following parameters were varied during testing:

- gossip fanout parameter;
- network packet loss probability;
- node crash events.

Metrics were collected directly from structured logs periodically produced by each node, including sent and received messages, membership states, and failure detector transitions.

5.2 Baseline Convergence

The baseline experiment verified correct dissemination and convergence under reliable network conditions. All nodes progressively exchanged membership information and eventually converged toward a consistent cluster view. The logs confirmed that all nodes maintained the states:

```
node-1 = ALIVE, node-2 = ALIVE, node-3 = ALIVE
```

throughout the execution.

5.3 Failure Detection Evaluation

A crash fault was simulated by terminating `node-3` during execution. The remaining nodes independently detected the failure through heartbeat timeout expiration. Table 1 summarizes the measured detection latencies.

Observer Node	SUSPECT Latency	DEAD Latency
node-1	6.16 s	12.18 s
node-2	6.08 s	12.10 s

Table 1: Failure detection latency after the crash of node-3.

The measured latencies closely match the configured timeout thresholds, demonstrating deterministic timeout-based failure detection.

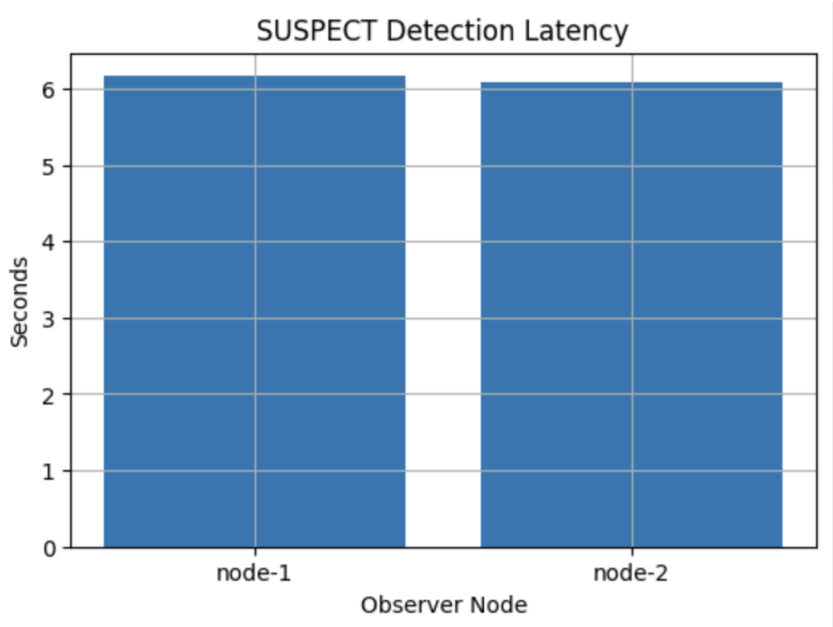

Figure 4: Latency required to transition node-3 into the SUSPECT state.

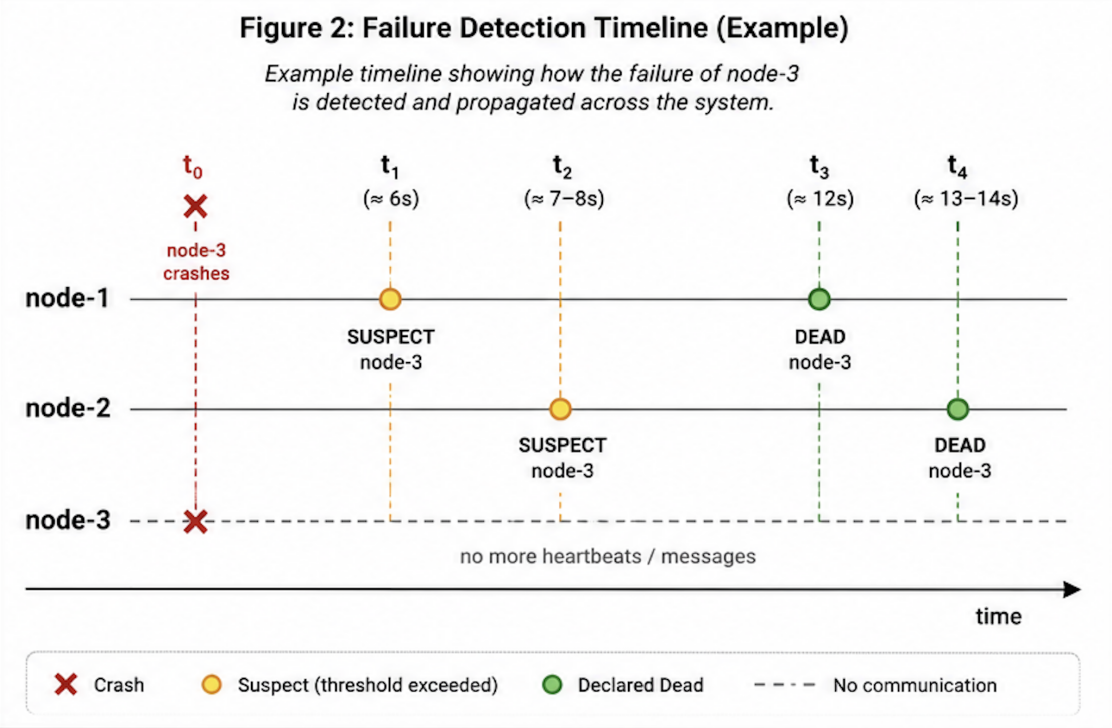

Figure 3: Failure detection and dissemination timeline after node crash.

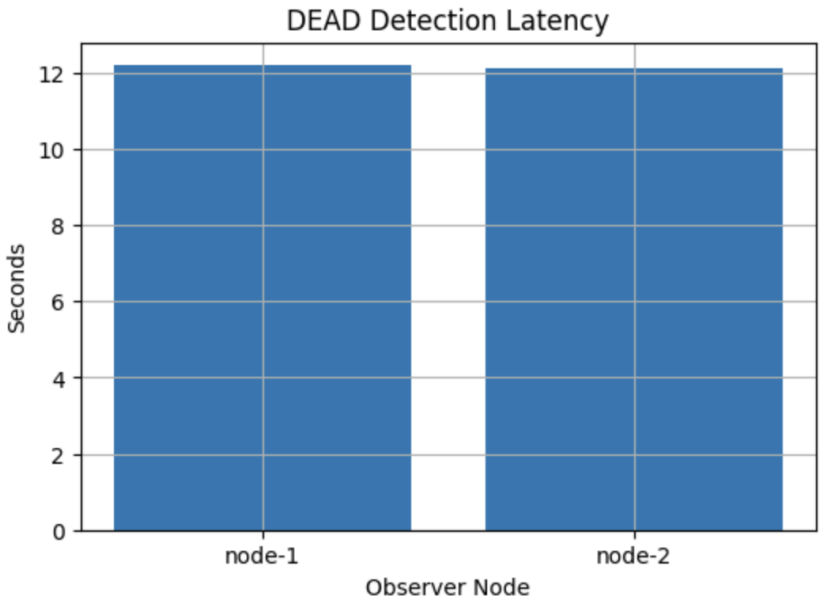

Figure 5: Latency required to transition node-3 into the DEAD state.

5.4 Packet Loss Robustness

The system was evaluated under unreliable network conditions using a packet loss probability of 30%. Despite the simulated losses, the cluster remained operational and eventually converged toward a consistent membership view. No node was permanently marked as failed. The experiment demonstrates the probabilistic robustness of epidemic dissemination: individual message losses are compensated by repeated gossip rounds.

5.5 Fanout Tradeoff Analysis

The fanout parameter directly influences dissemination redundancy and communication overhead.

Three experiments were conducted using:

- FANOUT = 1
- FANOUT = 2
- FANOUT = 3

Table 2 summarizes the observed message counts.

Fanout	Node-1 Sent	Node-2 Sent	Node-3 Sent
1	99	89	84
2	158	158	148
3	148	148	138

Table 2: Communication overhead under different fanout values.

The results show that increasing the fanout parameter increases communication overhead while improving dissemination redundancy. With a three-node cluster, however, each node only has two remote peers. Consequently, FANOUT=3 behaves similarly to FANOUT=2 because the effective fanout is bounded by the number of available peers. Experimental observations additionally showed that push-pull dissemination improves synchronization speed and reduces temporary membership divergence compared to pure push dissemination, particularly under unreliable network conditions. Bidirectional anti-entropy exchanges enabled nodes to reconcile membership information more rapidly during gossip rounds.

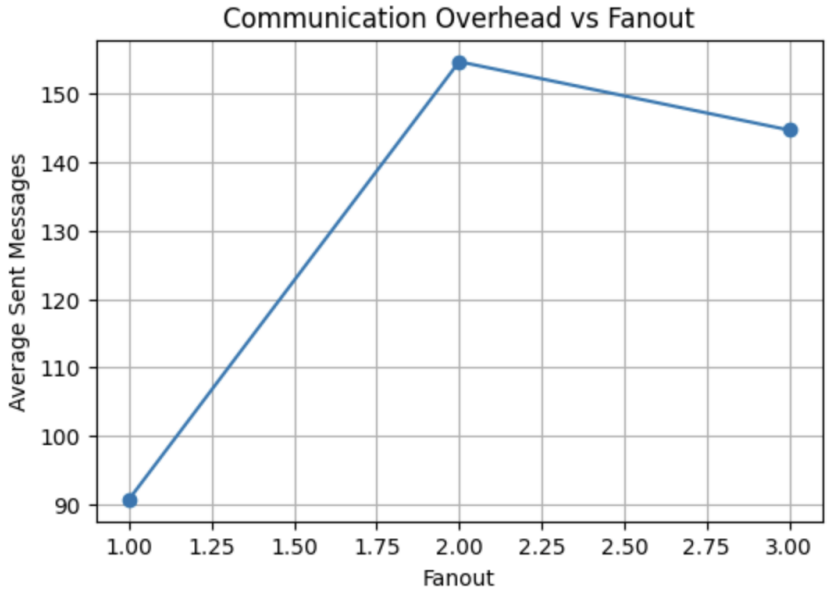

Figure 6: Average communication overhead as a function of fanout.

5.6 Discussion

The experiments confirm that the system exhibits the expected properties of gossip-based distributed monitoring systems:

- eventual consistency under asynchronous communication;
- decentralized failure detection;
- resilience to packet loss;
- scalable communication through bounded fanout dissemination.

The observed dissemination behavior follows the classical epidemic dissemination pattern described in the distributed systems literature.

6 Deployment

6.1 Requirements

The project requires the following software dependencies:

- Docker (version 20.10 or higher)
- Docker Compose v2 (version 2.x or higher)
- Python 3.9 or higher (for local execution without containers)

The system is platform-independent and has been successfully tested on macOS and Linux environments.

6.2 Repository Setup

The project repository can be cloned locally using:

```
git clone <repository-url>
cd gossip-distributed-monitoring
```

Dependencies for local execution can be installed through:

```
pip install -r requirements.txt
```

6.3 Containerized Deployment

The distributed cluster can be executed through Docker Compose.

To build and start the system:

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

To stop the cluster:

```
docker compose down
```

Docker Compose creates an isolated virtual network in which each node executes as an independent containerized process communicating through UDP sockets.

6.4 Docker Compose Design

The `docker-compose.yml` file defines each node as a separate service.

Each node is configured with:

- a unique `NODE_ID`;
- a dedicated `BIND_PORT`;
- a `PEERS` list containing known peer addresses in the format: `NODE_ID@HOST:PORT`;
- configurable gossip and failure detection parameters.

All services are connected to a shared Docker bridge network (`gossip-net`), enabling hostname-based peer communication.

A simplified example configuration is shown below:

```
services:
  node-1:
    build: .
    environment:
      NODE_ID: node-1
```

```
    BIND_PORT: 5001
    FANOUT: 2
    PEERS: >
      node-1@node-1:5001,
      node-2@node-2:5002,
      node-3@node-3:5003
```

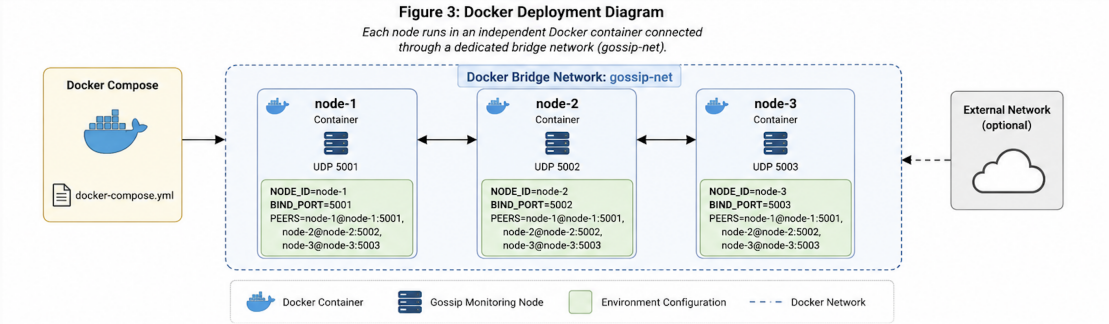

Figure 7: Containerized deployment architecture using Docker Compose.

6.5 Local Multi-Process Execution

For experimentation and validation purposes, nodes can also be executed locally as independent Python processes.

Example:

```
NODE_ID=node-1 \
BIND_PORT=5001 \
FANOUT=2 \
PEERS=node-1@127.0.0.1:5001,\
node-2@127.0.0.1:5002,\
node-3@127.0.0.1:5003 \
python3 -m src.main
```

Multiple terminals can be used to simulate a distributed cluster on a single machine.

6.6 Fault Injection Configuration

The system supports configurable network impairment simulation through environment variables.

For example, packet loss can be introduced using:

```
NET_DROP_RATE=0.3
```

Similarly, artificial communication delay can be configured through:

```
NET_DELAY_MIN_MS=50
NET_DELAY_MAX_MS=200
```

These parameters were extensively used during the experimental validation phase.

6.7 Expected Runtime Behaviour

After startup, nodes periodically exchange membership information through gossip dissemination.

The following logs are expected during normal execution:

```
[METRICS] node=node-1 sent=35 recv=17 suspect=0 dead=0

[MEMBERSHIP]
node-1 hb=19 inc=1 status=ALIVE
node-2 hb=12 inc=1 status=ALIVE
node-3 hb=11 inc=1 status=ALIVE
```

When a node is stopped intentionally:

```
[FD] node-3 -> SUSPECT
[FD] node-3 -> DEAD
```

The remaining nodes progressively propagate the updated failure information until the cluster converges toward a consistent membership state.

6.8 Reproducibility

All experiments presented in the Validation section are fully reproducible through the provided Docker Compose configuration and environment-variable-based parameterization.

This reproducibility was considered an important design goal in order to support controlled experimentation on dissemination dynamics, convergence latency, and fault-tolerance behavior.

7 User Guide

The system is designed to operate autonomously once deployed. User interaction is intentionally minimal and mainly focused on: deployment, observation, and fault-injection experimentation.

7.1 Starting the System

The distributed cluster can be started through Docker Compose:

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

After startup, nodes automatically begin periodic gossip dissemination, membership synchronization, and failure detection activities.

7.2 Observing Runtime Behaviour

Each node continuously emits structured logs containing:

- communication metrics;
- membership state updates;
- rumor dissemination events;
- failure detector transitions.

Logs can be monitored in real time through:

```
docker logs -f gossip-node-1
```

Typical runtime output includes:

```
[METRICS] node=node-1 sent=35 recv=17 suspect=0 dead=0

[MEMBERSHIP]
node-1 hb=19 inc=1 status=ALIVE
node-2 hb=12 inc=1 status=ALIVE
node-3 hb=11 inc=1 status=ALIVE
```

7.3 Simulating Node Failures

Crash failures can be simulated by stopping a running container:

```
docker stop gossip-node-3
```

The remaining nodes will eventually transition the failed node through:

$$\texttt{ALIVE} \rightarrow SUSPECT \rightarrow DEAD$$

according to the configured timeout thresholds.

The resulting transitions are visible in the logs:

```
[FD] node-3 -> SUSPECT
[FD] node-3 -> DEAD
```

7.4 Simulating Unreliable Networks

The system supports configurable network impairment simulation through environment variables.

Packet loss can be enabled using:

```
NET_DROP_RATE=0.3
```

Artificial communication delay can be introduced through:

```
NET_DELAY_MIN_MS=50
NET_DELAY_MAX_MS=200
```

These parameters affect dissemination latency, convergence speed, and failure detection stability.

7.5 Local Multi-Node Execution

For experimentation purposes, nodes can also be executed locally as independent Python processes:

```
NODE_ID=node-1 \
BIND_PORT=5001 \
FANOUT=2 \
PEERS=node-1@127.0.0.1:5001,\
node-2@127.0.0.1:5002,\
node-3@127.0.0.1:5003 \
python3 -m src.main
```

Running multiple terminals enables the simulation of a distributed environment on a single machine.

7.6 Gossip Arena

The optional *Gossip Arena* component enables controlled rumor dissemination experiments.

Rumors can be injected through the FastAPI endpoint exposed by a node:

```
curl -X POST http://localhost:8001/arena/inject \
  -H "Content-Type: application/json" \
  -d '{"rumor_id":"r42"}'
```

Injected rumors are progressively disseminated across the cluster through epidemic propagation and become observable in node logs.

This functionality was primarily used during experimental validation to study dissemination latency, propagation coverage, and convergence dynamics.

8 Self-evaluation

This project was entirely designed and implemented as an individual effort by Thomas Testa.

The development process involved the study, design, implementation, experimental validation, and documentation of a fully decentralized gossip-based monitoring system inspired by epidemic dissemination protocols and SWIM-like membership management techniques.

8.1 Strengths

- **Fully decentralized architecture**: the system successfully eliminates centralized coordination and single points of failure. Each node independently participates in dissemination, membership management and failure detection, closely reflecting the principles of real distributed monitoring systems.
- **Experimental validation**: the project includes reproducible experiments evaluating convergence behavior, communication overhead, packet-loss resilience, and failure detection latency. This significantly strengthens the engineering quality of the project.
- **Robustness under unreliable communication**: the system remains operational despite packet loss, delayed communication, and node crashes. Experimental results confirm the resilience properties typically associated with epidemic dissemination protocols.
- **Scalability-oriented communication model**: the use of bounded fanout dissemination ensures that communication overhead grows proportionally to the chosen fanout parameter rather than quadratically with cluster size.
- **Modular software architecture**: networking, dissemination, membership management, and failure detection are implemented as clearly separated components, improving maintainability and extensibility.
- **Reproducibility**: the entire system can be deployed and tested deterministically through Docker Compose and environment-based configuration, enabling controlled experimentation.

8.2 Weaknesses and Limitations

- **False positives in failure detection**: under severe packet loss or prolonged network delay, healthy nodes may be incorrectly marked as `SUSPECT` or `DEAD`. This limitation is intrinsic to timeout-based failure detection in asynchronous systems.
- **Static timeout configuration**: timeout thresholds are currently fixed and manually configured. Adaptive timeout estimation strategies, similar to those used in SWIM [2], would improve robustness under heterogeneous network conditions.
- **Parameter sensitivity**: dissemination quality depends strongly on parameters such as fanout, gossip interval, and timeout duration. Improper tuning may increase convergence latency or communication overhead.
- **Limited scalability testing**: experiments were executed on relatively small clusters due to local hardware constraints. Large-scale deployments would require additional evaluation regarding bandwidth usage and dissemination saturation effects.

- **Lack of security mechanisms**: the system assumes a cooperative, non-malicious environment. No authentication, encryption, or Byzantine fault tolerance mechanisms are implemented.
- **No persistent storage**: membership information is maintained entirely in-memory. Node restarts therefore lose all local state and require re-synchronization through gossip dissemination.

8.3 Lessons Learned

The project provided practical experience with several fundamental distributed systems concepts, including:

- epidemic dissemination protocols;
- eventual consistency;
- decentralized failure detection;
- asynchronous communication;
- fault tolerance under unreliable networks;
- tradeoffs between scalability, redundancy, and convergence speed.

Particular attention was devoted to understanding how probabilistic dissemination can compensate for unreliable communication without relying on centralized coordination or reliable transport protocols.

8.4 Overall Assessment

Overall, the project successfully achieves its primary objectives and demonstrates a solid understanding of distributed systems principles.

The resulting system represents a realistic experimental platform for studying gossip-based dissemination, decentralized membership management, and failure detection dynamics in asynchronous environments.

9 Future Works

Although the current implementation successfully demonstrates decentralized membership dissemination and failure detection through gossip protocols, several extensions could further improve the system in terms of robustness, scalability, realism, and research value.

- **Adaptive failure detection**: introduce adaptive timeout mechanisms based on observed network conditions, such as ϕ-accrual failure detectors [4]. This would reduce false positives while preserving low detection latency under heterogeneous network conditions.

- **SWIM-style indirect probing**: implement the indirect probing mechanism proposed by SWIM [2], allowing nodes to verify suspected failures through intermediary peers. This would improve resilience against temporary packet loss and asymmetric communication failures.
- **Comparative dissemination benchmarking**: future work could extend the current implementation with automated benchmarking tools to systematically compare push and push-pull dissemination strategies under different network impairment conditions, cluster sizes, and fanout configurations.
- **Advanced peer sampling**: integrate more sophisticated peer sampling protocols such as HyParView [6] to improve overlay connectivity, dissemination robustness, and scalability under dynamic churn conditions.
- **Large-scale deployment**: evaluate the system on substantially larger clusters (50–200+ nodes) through cloud-based infrastructures, distributed virtual machines, or network simulation environments.
- **Dynamic peer discovery**: remove the requirement for static bootstrap peer lists and introduce decentralized service discovery mechanisms to support more realistic dynamic deployments.
- **Security and trust mechanisms**: add message authentication, integrity validation, and protection against malicious or Byzantine nodes. This would be necessary for deployment in untrusted environments.
- **Persistent membership storage**: introduce optional persistent state replication so that nodes can recover membership information after crashes or restarts without requiring full re-synchronization.
- **Adaptive dissemination policies**: dynamically tune fanout, gossip interval, and timeout parameters according to observed network conditions and dissemination quality metrics.
- **Visualization and observability**: develop a real-time dashboard for monitoring cluster topology evolution, rumor propagation, failure dissemination, and communication metrics. Automatic chart generation from experiment logs could further improve reproducibility and analysis.
- **Kubernetes-native deployment**: extend the deployment model toward orchestration platforms such as Kubernetes, enabling experiments involving elastic scaling, container rescheduling, and cloud-native failure scenarios.
- **CRDT-based replicated state**: future versions could integrate Conflict-free Replicated Data Types (CRDTs) to support stronger replicated shared-state semantics while preserving decentralized consistency guarantees.

Overall, the project provides a solid foundation for further experimentation on epidemic dissemination protocols and decentralized monitoring systems, with several opportunities for both engineering improvements and research-oriented extensions.

References

[1] Eric A. Brewer. Towards robust distributed systems. In *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 7. ACM, 2000.

[2] Abhinandan Das, Indranil Gupta, and Ashish Motivala. SWIM: Scalable weakly-consistent infection-style process group membership protocol. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 303–312. IEEE, 2002.

[3] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. *ACM SIGOPS Operating Systems Review*, 22(1):8–32, 1988.

[4] Naohiro Hayashibara, Xavier Defago, Rami Yared, and Takuya Katayama. The ϕ accrual failure detector. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 66–78. IEEE, 2004.

[5] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems*, 25(3):8–es, 2007.

[6] João Leitão, José Pereira, and Luís Rodrigues. HyParView: A membership protocol for reliable gossip-based broadcast. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 419–429. IEEE, 2007.