NoSql-cli-emulator

Distributed systems
2024-2025

Project goal

The project goal is to develop a lightweight, distributed NoSQL emulator that implements sharding and replication using a peer-to-peer (P2P) architecture. The system will follow the BASE (Basically Available, Soft state, Eventually consistent) approach, ensuring high availability and reasonable fault tolerance while prioritizing low latency for read and write operations. It's a cli application, interaction will be based on scripts or command line commands. his project explores a minimal, peer-to-peer approach to NoSQL emulation, focusing on:

1. **Synchronization** – Construct a mechanism that allows nodes to be up-to-date, with low effort and good efficiency;

2. **Replication** – Maintaining multiple copies of each data to ensure availability and durability;

3. **BASE Semantics** – Embracing eventual consistency to optimize for availability and partition tolerance, while keeping the system's state "soft" and reconciliation asynchronous.

The P2P design removes the need for a central coordinator, simplifying deployment and demonstrating fault-tolerant behaviors in a distributed setting. This choices will be discussed in depth in the next chapters.

Requirements

- Functional requirements;
  - Data management → the system must allow storing, retrieving, modifying and deleting data;
  - Data replication → The system must implement a gossip protocol mechanism that allows data replication between nodes.;
  - Heartbeat and node monitoring → to monitor nodes health;
  - Service discovery → Every peer should be able to find other nodes, by using a service discovery mechanism;
  - Data synchronization;
  - sharding;
  - Roles and permissions available through api key.
- Non-functional requirements;
  - Scalability → The system must be able to scale horizontally, meaning it should handle an increasing number of nodes and data volume without significant performance degradation;
  - Fault tolerance → The system should be resilient to node failures and network issues, ensuring continued operation even in the presence of failures;
  - Eventual consistency and soft state.

Technologies used

- python;
  - fastAPI;
  - Uvicorn;
  - Pytest;
  - Httpx;
  - pytest-mock;
  - network;
  - xxhash.
- Docker;
- Postman.

Design: Architecture

The application follows a peer-to-peer (P2P) architecture, where each node is both a client and a server. Since this project aims to emulate nosql replication and synchronization mechanism, the choice was between master-slave and P2P, since these two architectures are the ones used in specific implementations. P2P was preferred, since:

- in master slave, all write operations must pass trough the master node, creating a single point of failure and congestion, that can limit the performance, when the write load is pretty high. In this project, since low latency is a key goal, it's very important to maintain an optimal scalability with a high write demand.

- it's decentralized, and the fact that each node can operate independently allows to implement BASE mechanism in a simpler way, and it's easier to handle failures;

- in general, better support for soft state and eventual consistency, allowing the implementation of some specific patterns (like gossip protocol).

Design: Infrastructure

- The system consists exclusively of peer nodes, each of which acts simultaneously as both a client and a server. Each node includes a local in-memory key-value store, an api layer, a gossip communication module and an heartbeat module. So basically, the number of components depends on the number of active peers.

- Peer nodes are distributed across different machines, and they may reside on the same local network (docker compose testing) or may be geographically distributed. This infrastructure does not have specific restrictions regarding the physical or network proximity of the nodes. Communication between nodes occurs through standard HTTP/REST APIs.

- Each peer can find others, since a decentralized service discovery mechanism was implemented. It's based on a non-blocking UDP broadcast loop, in which messages containing node id and address are included. Each peer memorizes the discovered addresses locally.

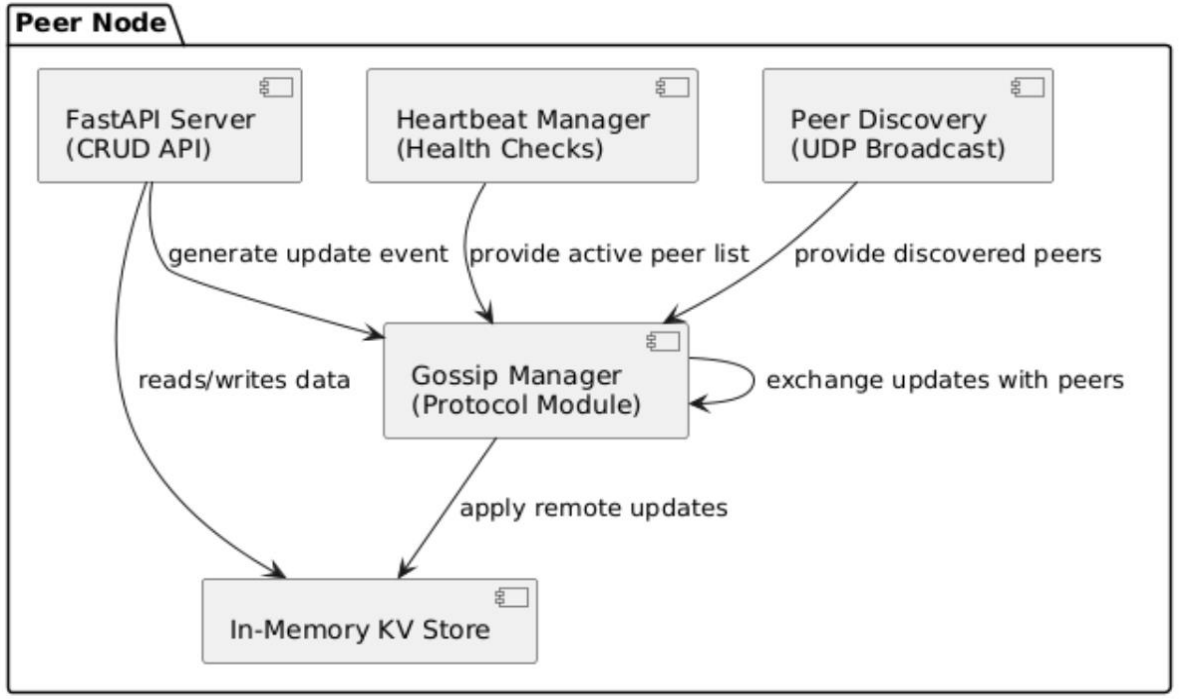
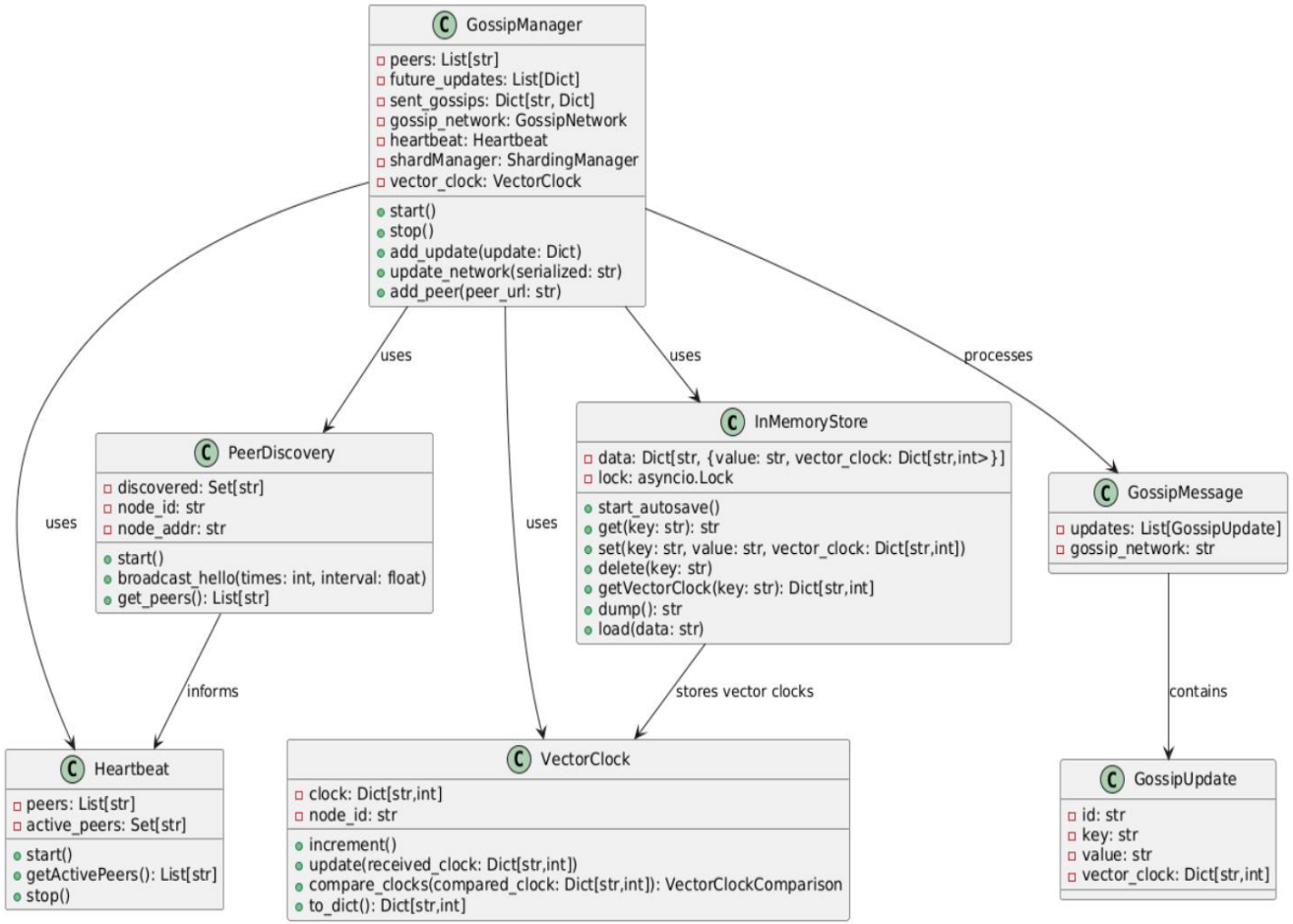


figure 2. component diagram, explaining the logic behind the structure of a single peer.

Design: Modelling



Domain Entities

- **KeyValueStore**
  A logical container for all key–value pairs. Under the hood, data is sharded and replicated across multiple nodes.
- **Node**
  A peer in the P2P network. Each runs as a Docker container hosting a FastAPI server.
- **VectorClock**
  Tracks causal history of updates for each key to resolve concurrent writes.
- **GossipMessage**
  Carries updates between nodes using HTTP POST `/gossip`.
- **HeartbeatMessage**
  A simple HTTP GET to `/heartbeat` for liveness checks.
- **PeerUpdate**
  Broadcast via UDP to discover new or returning peers.

Domain Entities Mapping to Infrastructural Components

Each domain concept has its concrete implementation, such that:

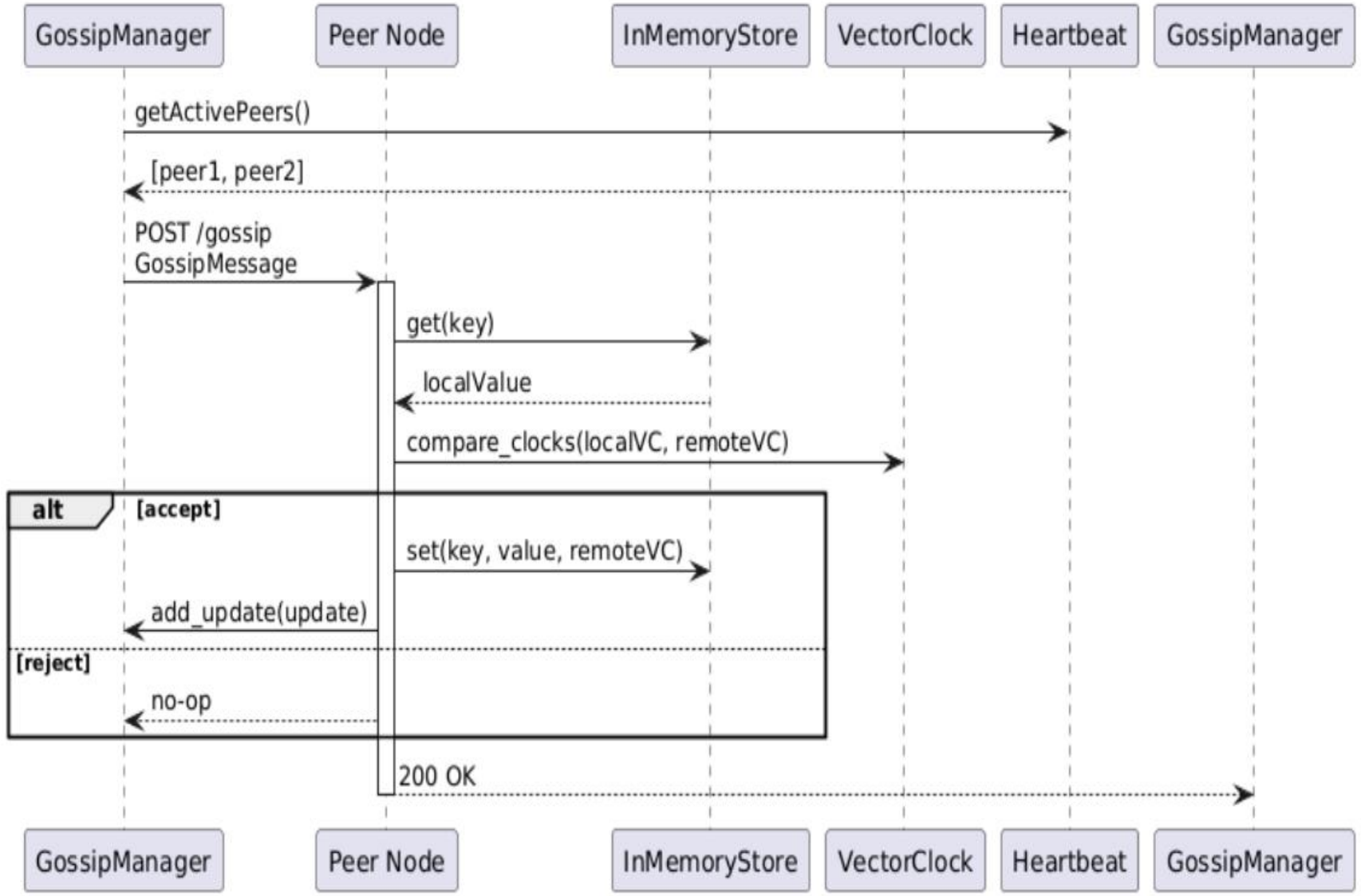
Concept	Implementation
Key–Value Store	In-memory dict + periodic disk flush
Node	Docker container + FastAPI server
VectorClock	Custom Python object per key
GossipMessage	POST payload on `/gossip` endpoint
Heartbeat	GET request on `/heartbeat` endpoint
Peer Discovery	UDP broadcast socket

Data replication: Gossip protocol

The gossip protocol is a decentralized replication mechanism inspired by how information spreads in social networks. It basically disseminate updates without a central coordinator, so it fits well a P2P architecture.

Each node periodically selects a subset of peers with which to exchange state, expecting that, in a predetermined time frame, the system will eventually be consistent. The peer selection can be tuned: to reduce traffic, nodes typically choose only a small random subset of peers each round. Gossip protocol gives "for free":

- failure handling (if a peer is unreachable, its updates will be picked up in a later round);
- Scalability and decentralization



Conflict resolution: lww

Conflict resolution

Conflict resolution handles the conflict using the last write wins (LWW) mechanism, so depending on the vector clocks, it changes its state (ACCEPT and REJECT)

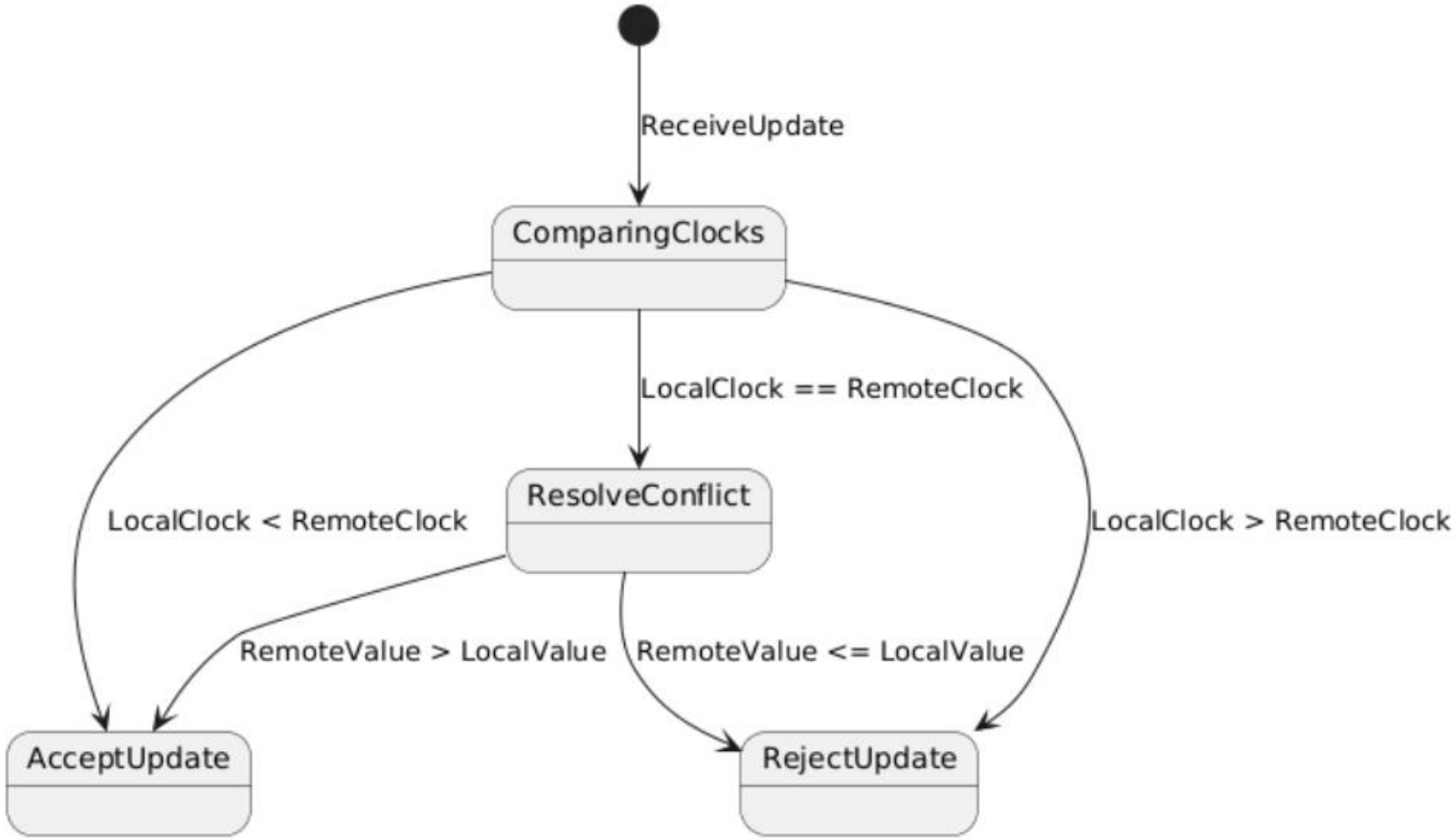


figure 5.4. state diagram, displaying conflict resolution.

Peer discovery

- **Description**: Every peer should be able to find other nodes, by using a service discovery mechanism. The implementation needs to be made with UDP protocol or broadcast messages, since a centralized approach will be avoided.

- **Acceptance Criteria**:

  - system should recognize and include in the p2p network, new added nodes;

  - system should work even without static definition of nodes;

  - service discovery should always be up, and decentralized.

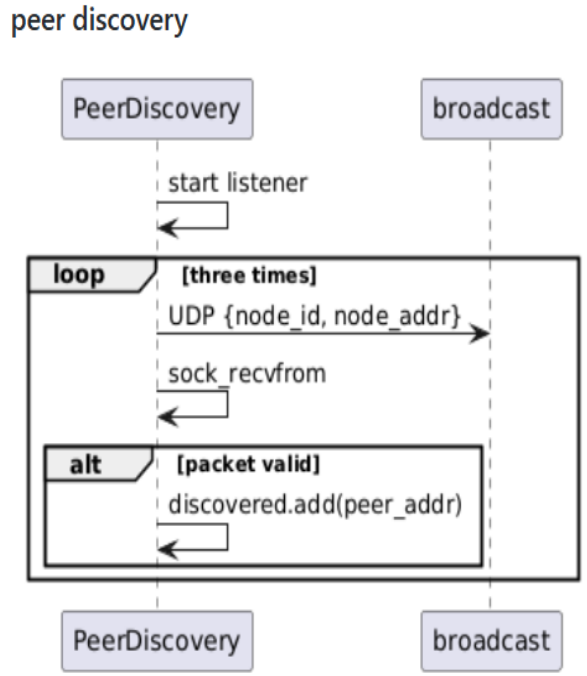


figure 4.4. sequence diagram displaying how service discovery works, using udp broadcast mechanism.

Sharding

This project also features sharding, and it can be activated through an environment variable.

The chosen sharding mechanism is Hash sharding, and it allows to redistribute data and reduce the computation load that each node has to deal with.

While sharding is enabled, data should be saved accordingly to the hash function.

```python
class ShardingManager:
    Qodo Gen: Options | Test this method
    def __init__(self):
        pass

    Qodo Gen: Options | Test this method
    def getHashedShardNumber(self, key: str, nodes_number:int = None) -> int:
        if nodes_number is None:
            nodes_number = len(os.getenv("ALL_PEERS", "").split(","))

        key_hash = xxhash.xxh64(key.encode()).intdigest()
        destination_node = key_hash % nodes_number
        return destination_node      Alessandro Pioggia, 4 weeks ago • chore: rem
```

```python
async def set(self, key, value, vector_clock: VectorClock = None):
    if self.shardManager is not None and self.shardNumber is not None:
        shardNumber:int = os.getenv("SHARD_ID", 0)
        correct_node = self.shardManager.getHashedShardNumber(key)
        if shardNumber != correct_node:
            return
```

Fault tolerance : Data replication

Data replication

- The system uses a gossip protocol to replicate updates between nodes and to synchronize data.
- When a key-value pair is updated on one node, the update is propagated to other nodes in the network. The propagation also includes the gossip network and the vector clocks, in order to have better performance, avoiding useless messages to be sent.
- Replication in this case ensures eventual consistency and soft state, meaning all nodes will converge to the same state over time and it’s also available.

Fault tolerance : Recovery

Node Failure

- if a node becomes unresponsive, it's up to the heartbeat mechanism to detect the node and to remove it from the active peer list. The gossip protocol and eventual consistency still allows users to access data.

Network partitions

- during a network partition, nodes in different partitions continue to operate independently. Once the partition is resolved, the gossip protocol synchronizes the data between partition.

Data recovery and reconciliation mechanism

- when a failed node rejoins the network, it receives missed updates that has been stored in other peers, and that's triggered initially by the heartbeat mechanism.

Conclusion

Personal thoughts

During the realization of this project I could experiment and work on a lot of aspects about distributed systems, such as:

- replication;
- network communication;
- handling failures;
- dockerization;
- eventual consistency, soft state;
- clocks synchronization. This work, helped me understand better the practical part of this course, and in some cases it was pretty challenging to get an acceptable result. The most difficult aspect was the gossip protocol implementation, since it handled failure resolution and replication and maybe because it was the first time I implemented it. To conclude, I'm satisfied with the process that led to the realization of the project, since it helped me to understand the subject.

Future works

It could be interesting to:

- work on performance;
- try out other network protocols;
- add advanced database features (such as search, filtering and other functionalities).
