SHOPPING LISTS ON THE CLOUD

Large Scale Distributed Systems Project

Group 14:

Luís Vieira Relvas - up202108661

Rodrigo Campos Rodrigues - up202108847

Wallen Marcos Ribeiro - up202109260

U.PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

INTRODUCTION

Our Project is a Local-First Shopping List Application. This application allows users to:

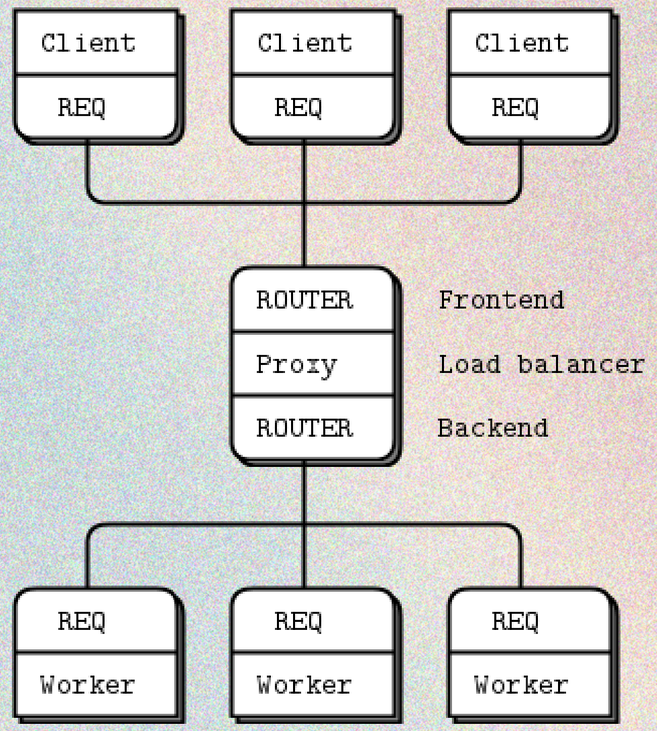
- **Create, edit, and delete shopping lists**
- **Add, remove, and modify items within lists**
- **Share lists with others for collaborative shopping**

The main technologies used, were the following:

ØMQ

COMMUNICATION PATTERNS

Load Balancing Broker where both clients and server are REQ. The clients “speak” with the Frontend (ROUTER) and the workers with the Backend (ROUTER).



Source of the image

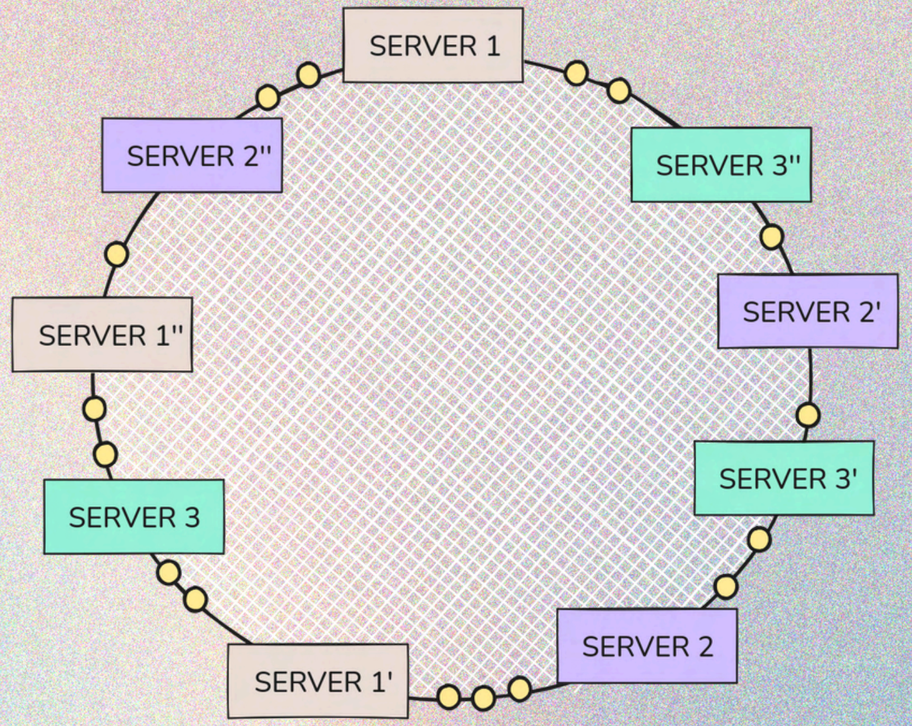
CLIENT

Each client maintains a local replica of shopping lists in JSON files. This enables offline-first functionality, allowing users to interact with their shopping lists even without a network connection.

Client is not aware of the internal details of the distributed system, such as data partitioning or node responsibilities. Instead, it routes its requests through a generic load balancer. Allowing it to remain lightweight and independent of system-specific logic.

LOAD BALANCER

It features a HashRing class that implements consistent hashing to efficiently distribute client requests across a set of worker nodes. By utilizing virtual nodes, the system ensures a more even distribution of traffic, preventing certain workers from becoming overloaded while others remain underutilized.



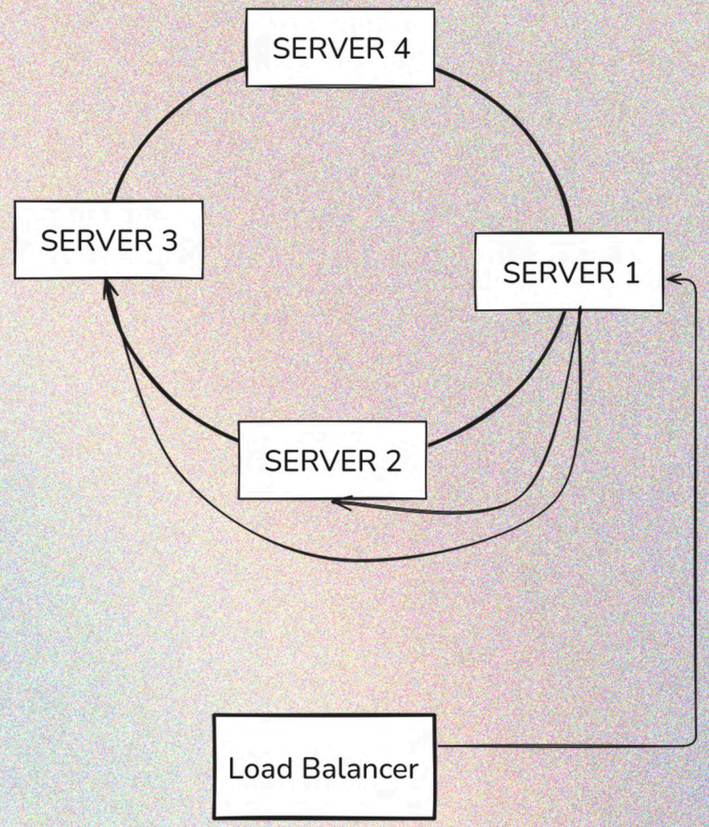
LOAD BALANCER

For worker health, a separate thread regularly checks if workers are active by sending a "PING". Unresponsive workers are dynamically removed from the system and the hashing ring adjusts task assignments to the remaining workers.

Client requests are received at the frontend (via a ROUTER socket) and forwarded to a designated worker in the backend. The frontend queries the HashRing to identify the coordinator worker.

WORKER/SERVERS

Workers receive list data and a preference list for replication from the Load Balancer. This ensures data availability in case of failures and is handled in separate threads to ensure that it doesn't block the worker's main execution flow.



CONFLICTS AND FAULT TOLERANCE

When conflicts arise between local and server states (e.g., simultaneous updates), the client merges CRDT states with the server to ensure a consistent and accurate representation of the data.

Server synchronization can be achieved when a client sends a request, such as a read or write operation, it triggers synchronization mechanisms among servers.

In the case a worker is down, due to consistent hashing another worker is reallocated making sure replication is successfully executed to at least N nodes in the preference list.

CRDT IMPLEMENTATION

- **PNCounter**
  - Representation:
    - {”item_id”: {”inc”:0, “dec”:0}}
  - Each client has a PNCounter instance for each list they have access to;
  - Each server stores a PNCounter instance for each list they have received until a certain moment.

CRDT IMPLEMENTATION

- **ORMap (for adding and removing items)**
  - Representation:
    - {”items”: {”item_id”: [dot1, dot2, dotn]}, “tombstones”: {{”item_id”: [dot1, dot2, dotn]}}, “dot_context”: [dot1, dot2, dotn]}
  - Each client has an ORMap instance for each list they have access to;
  - Each server stores an ORMap instance for each list they have received until a certain moment;

CRDT IMPLEMENTATION

- **ORMap (for adding and removing lists)**
  - Representation:
    - {”items”: {”list_id”: [dot1, dot2, dotn]}, “tombstones”: {{”list_id”: [dot1, dot2, dotn]}}, “dot_context”: [dot1, dot2, dotn]}
  - Each client has an ORMap instance with all the lists they have access to;
  - Each server stores an ORMap instance with all the lists they have received until a certain moment.

CRDT IMPLEMENTATION

- **ORMap**
  - We opted to add the “tombstones” data structure to our ORMap representation to reduce computational effort. Otherwise, we would have to always go through the “items” and the “dot_context” structures to check for removed items.

MAIN CHALLENGES

- Understanding Key Principles: Understanding concepts like consistent hashing, unique tags for conflict resolution, eventual consistency, and replication.
- Coordinating effectively as a group to divide tasks like server-side architecture, and CRDT implementation.

FUTURE WORK

- Currently, when a client deletes a list, it is removed from the client's database, but the servers keep the list on theirs. We chose to keep it this way, because we would like to expand the functionality of our application and allow users to restore previously deleted lists.

- In the future, we also plan to implement a gossip protocol to merge server data periodically to ensure consistency across all of them.

THANK YOU!