U.PORTO
FEUP FACULTY OF ENGINEERING
UNIVERSITY OF PORTO

Shopping Lists on the Cloud

Large Scale Distributed Systems

Rita Leite - 202105309
Sofia Moura - 201907201
Tiago Azevedo - 202108699

Context

- This project will explore creating a local shopping list app.
- The app has code that runs on the user's device and can **persist data locally**, as well as having a **cloud component** to share data between users and provide backup storage.
- Users can create a new shopping list through the user interface. After creation and until a list is deleted, it exists under a unique ID that can be shared with other users. Users who know this ID should always have permission to add and delete items from the list.
- Each item has a target quantity associated with it that decreases as items are purchased.
- Targeting millions of users, the application has been carefully designed, particularly the cloud-side architecture to **avoid bottlenecks** in data access.

Technologies

- **sqlite3** - used to manage the databases of the servers and the clients.
- **zmq** - used for the comunication between servers, brokers and clients.
- **hashlib** - used to generate an hash for the servers' id and the lists' id.
- **uuid** - used to generate unique identifiers for the shopping lists.

Main Design Choices

- The application uses a local-first approach, ensuring that the data remains locally persistent on users' devices, which provides and agile and responsive experience on experiences of limited connectivity.
- List interactions occur in the local component, allowing users to update data even when they are offline. They can perform the following actions
  - Creation of shopping lists
  - Access shopping lists that have already been created (even if the ones not created by them)
  - View the items, and their quantities, that are on the shopping lists
  - Add new items to shopping lists
  - Remove items from shopping lists

Main Design Choices

- **Servers are distributed across a hash ring using virtual nodes.**
  - Each server corresponds to three virtual servers, represented by unique hash values (arranged in an ordered list). When handling requests for a specific list, the hash of the list ID is calculated, and the request is forwarded to the first server whose hash is greater than the list's hash. This way, each server only needs to be aware of a portion of the data (reducing the resource overhead) and the use of virtual nodes helps evenly distribute the load across the hash ring (improving the system's scalability and fault tolerance).
- **The broker knows the hash ring configuration and acts as an intermediary between the client and the server.**
  - The broker is aware of the hash ring and is responsible for redirecting the client’s message to the appropriate server, provided no errors occur. Without this knowledge, the broker could forward the message to a random server, but this would lead to unnecessary message exchanges. If the message was sent to the wrong server, it would need to be forwarded again to the correct one, which would increase communication overhead. By acting as an intermediary between the client and the server, the broker simplifies the client's logic.

Database Schema

- The database is made up of two tables, one with customer information and the other with shopping list information.
- The shopping list can only be deleted by its owner, which is why it has the "deleted" attribute. The crdt is stored directly in the database, in the list attribute called "crdt".

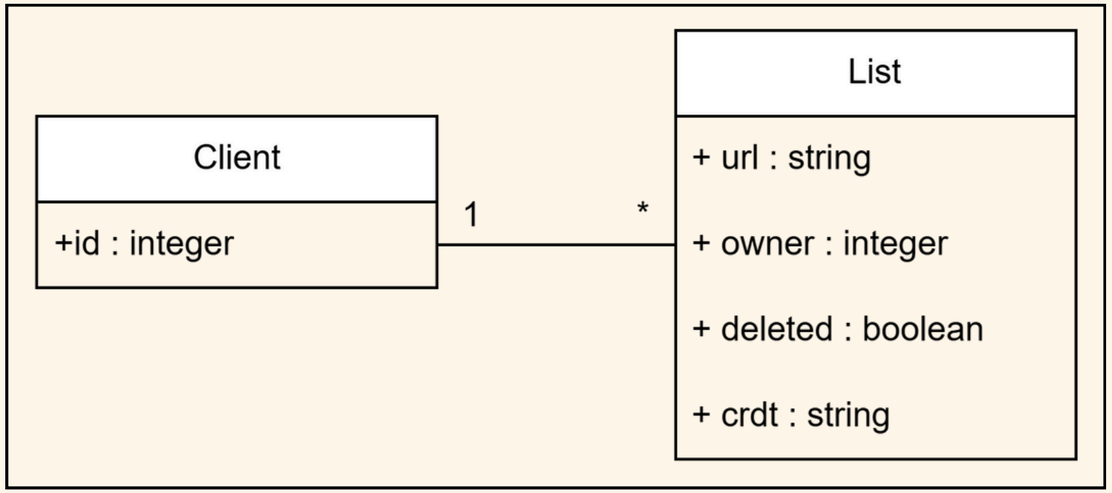

Figure 1 - Database Schema

Conflict-free Replicated Data Types

For the CRDT implementation, we use an AWMap to associate each shopping list item with a CCounter, along with a DotContext to track causality.

- **DotContext**
  - Tracks causality by mapping the most recent known version of each replica, ensuring changes can be correctly synchronized across nodes.
- **CCounter**
  - A CCounter is a distributed causal counter leveraging DotContext to manage context. Replicas increment or decrement the counter by updating its last known value, storing the new value under a new "dot", and removing the previous one. The total counter value is the sum of all replicas' counters.
  - We chose it for its support of increments and decrements and its efficiency in storing and processing by maintaining only one version per replica.
- **AWMap**
  - The AWMap is an excellent solution for managing multiple items in a distributed environment, handling concurrent changes smoothly while ensuring consistency and avoiding conflicts.
  - We chose it for its fast access to the values of each item.

Comunication Architecture

The client sends requests to the brokers, and they send replies to the clients. The broker sends requests to the servers, and they send replies to the broker.

- **The client's local database is periodically updated.**
  - The client's shopping list is periodically updated in the local database, ensuring that any changes are saved even when they are offline.

- **The client periodically synchronizes their shopping list with the server.**
  - The client sends a message containing its CRDT to one of the brokers (randomly chosen) at regular intervals and waits for a response for a specified amount of time (to avoid waiting indefinitely). This approach implements the polling technique.

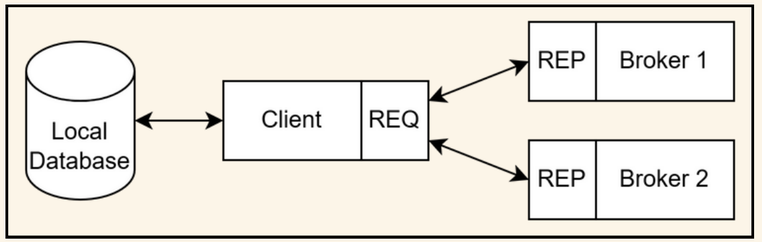

Figure 2 - Comunication Client - Brokers

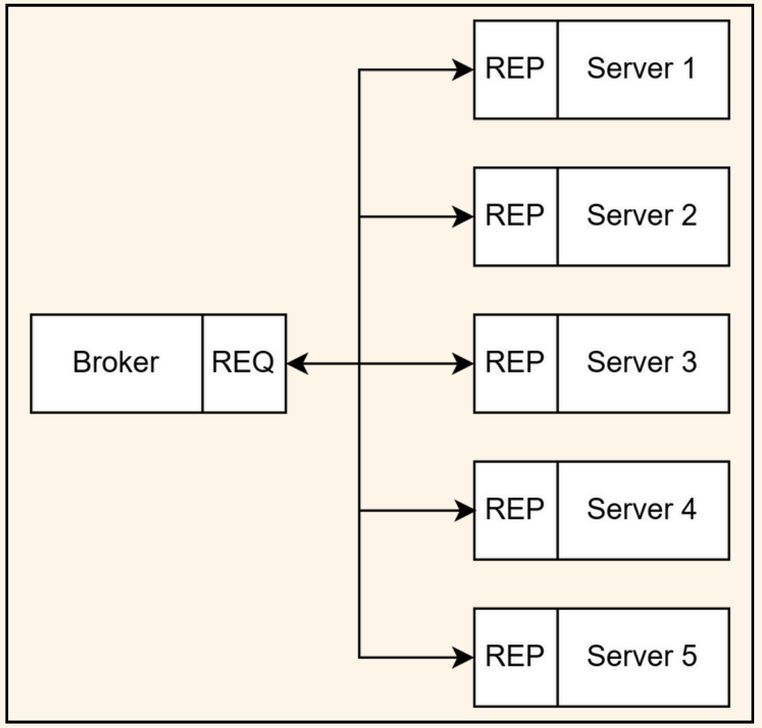

Figure 3 - Comunication Broker - Servers

Comunication Architecture

- **Virtual Nodes**
  - Due to the use of virtual nodes, the neighbors of a server may differ depending on the virtual node. Additionally, it is important to note that the virtual node neighbors have to corresponde to different “physical” servers.

- **Reads and Writes**
  - When a server receives a message to update a list, it must read the list from its neighbors (in this case, two servers) and write the update to those same neighbors.

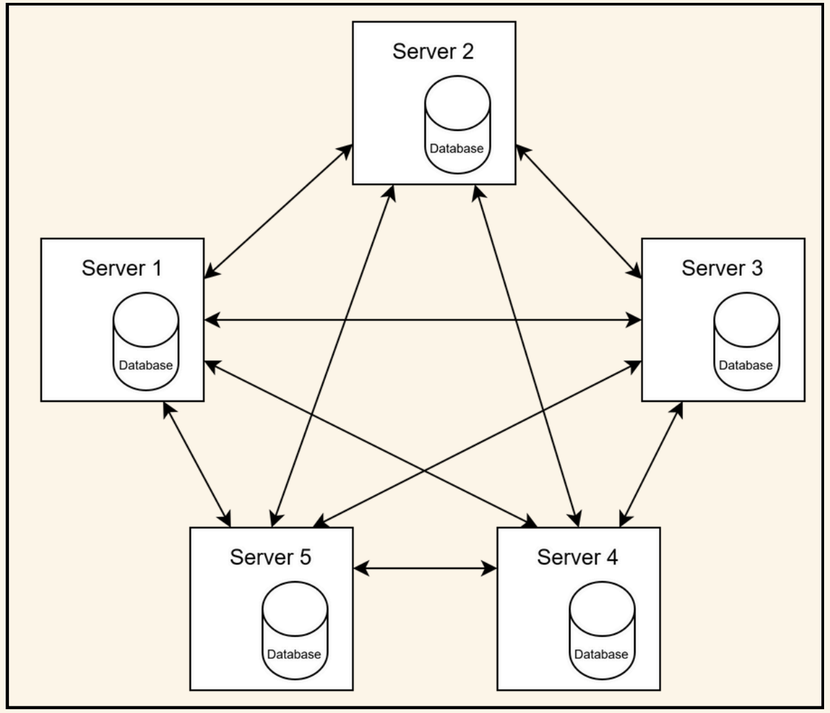

Figure 4 - Comunication Server - Server

- **Server Failures**
  - In the event of server failures, if a server cannot write to one of its neighbors, the update message is sent to the next available server, ensuring the update is propagated to servers. The server that processes the update on behalf of the failed server is responsible for attempting to forward the message to the failed server as soon as it becomes available again.