U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Local First Application

Shopping Lists on the Cloud

SDLE 23/24

T7G04
- António Ferreira (up202004735)
- João Maldonado (up202004244)
- Sérgio Carvalhais (up202007544)
- Tomás Gomes (up202004393)

Problem Description

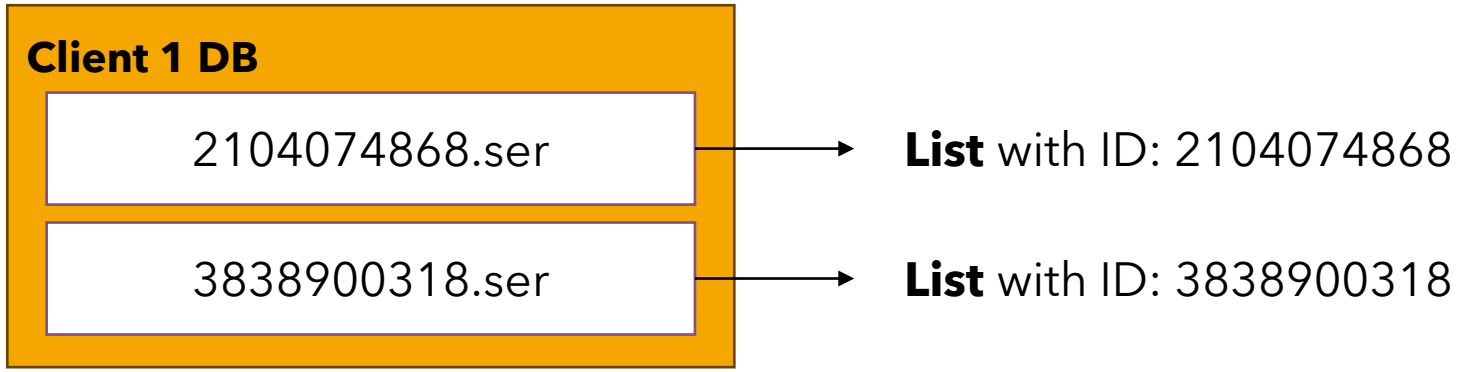
The main objective of this project is to create a **local-first shopping list platform**, that is designed to provide users with the ability to create, share and modify shopping lists. To achieve this, we adopted a client-server approach:

- The **client side** of the application provides an interface for the user to manage shopping lists. Users are also able to access and edit shared lists with a unique identifier (UID).
- The **cloud side** serves as the central repository for all shopping lists data, ensuring data persistence and synchronization across devices.

Also, ensuring consistency and resolving conflicts in data updates is crucial when multiple clients concurrently modify data. Therefore, our application not only needs to **guarantee local-first capabilities**, but also **ensure high availability and data persistence**.

Technical Solution
Client Side

For the client side of the application, we focused on guaranteeing a local first experience. To achieve this, **each client is given a database**. In the case of our application, we chose to store each created list in a different file, as a Java Serializable Object **(.ser files**). To distinguish the files in the database, we opted to use the ID of each list as the name of the file.



Technical Solution
Client Side

Regarding the ID of each list, we opted to use a random Long number, hashed with the **Murmur Hash** algorithm. More specifically, we utilized the x32 version, which yields a **32-bit hash value**.

The reasons behind our choice in this hash function was due to its efficient performance, simplicity and scalability (works well with data of many sizes). Also, considering that the main job of this function is to identify and store shopping lists, the fact that it's **Non-Cryptographic** doesn't stand as a disadvantage for its use.

Long Random Number → MurmurHash_x32() → **Hashed ID**

Technical Solution
Client Side

Now focusing on the features, that we implemented on this side of the application, a user can do the following:

1. **Create New List** – creates new list and stores it locally.
2. **Get List** – retrieves from the database the list with the ID chosen by the user.
3. **Delete List** – deletes the list locally. Also, the user can choose if they want to also delete the list remotely.
4. **Push List** – pushes the current state of a list chosen by the user to the cloud, for it to be stored.
5. **Pull List** – pulls from the cloud a list chosen by the user and stores it locally.

Technical Solution
Client Side

Whenever a user creates, retrieves or pulls a list, they will be given the opportunity to edit the list. More specifically, the user will be able to add and delete items from the list. To do this, the client will input the **name of the item** and its **quantity**. If the **item already exists**, the **quantity will be updated**. If the **item doesn't exist on the list**, it will be **created** with the **desired quantity**.

Technical Solution
Load Balancer

In our application, the load balancer is represented by our **Server Manager**, a class that is not only responsible for being the "middle-man" in the communication between the clients and the servers, but also for managing the process of **Consistent Hashing**. We decided to implement this distributed hashing technique as it ensures that our shopping lists are being loaded evenly through out the many servers we have available in the cloud, allowing for a more efficient data retrieval.

In the Server Manager, we'll have the **hash ring**, a virtual ring structure where each server, represented by a **hash number** (obtained by hashing the IP address of the server, with the Murmur Hash function), will be placed accordingly. The information regarding a server and its hash identifier will be stored in a **Server Table, a map where the values are the servers' IP addresses, and the keys are the hashed IP addresses.**

Technical Solution
Load Balancer

When the client pushes a list to the cloud, the Server Manager will be responsible for determining **in which server the list should be stored**. In our implemented Consistent Hashing mechanism, each server will only store the lists whose ID fall in the range between the server's hash number identifier and the hash number of its predecessor.

Example:

Range of Server 1 (4071703144): **]2412085335, 4071703144]**

Server 2 Server 1

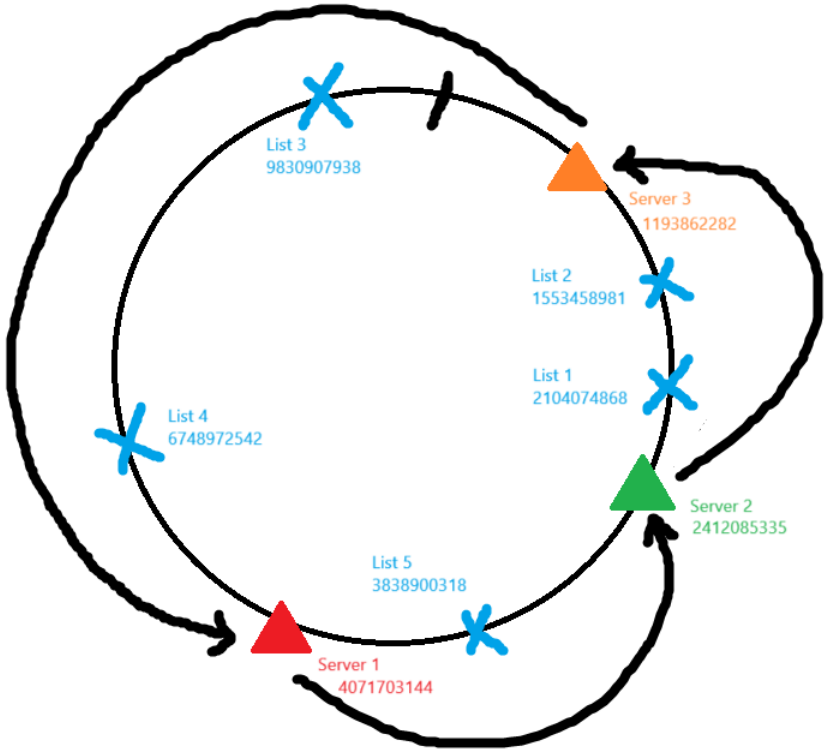


Fig. 1 – Consistent Hashing

Technical Solution
Load Balancer

In addition to Consistent Hashing, we also implemented **Replication** of the servers. In other words, instead of storing a list only in one server, **replicas of the same list will be stored in other servers**. This mechanism is essential for **maintaining high availability** of the application.

Therefore, in the Server Manager class, we introduced a **redundancy degree**, which is the number of servers that a list should be stored in. In this project, we implemented so that **each list is stored in its respective server and in its (redundancy degree – 1) predecessors** in the ring. For instance, if we consider a redundancy degree of 3, then a list will be stored in 3 servers (1 server + 2 predecessor servers).

Note:

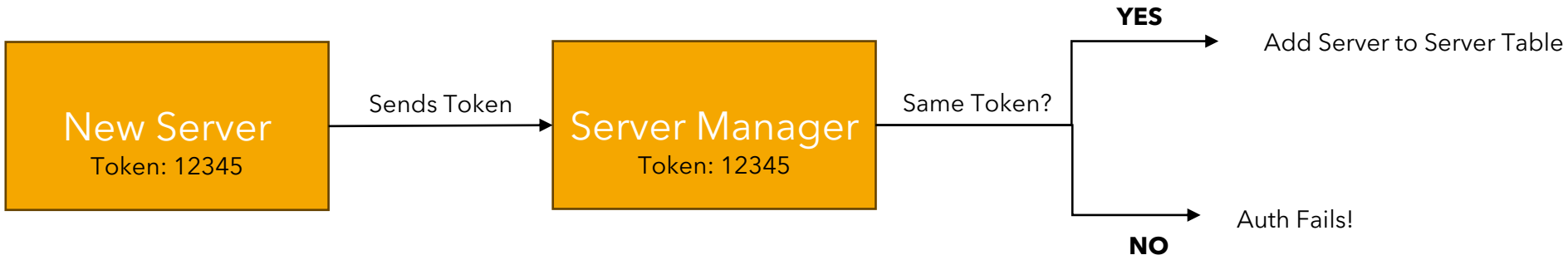
In the cases where the redundancy degree is higher than the number of available servers, then the lists pushed to the cloud will be stored in all the servers.

Technical Solution
Server

Similarly to the clients, each server in our application will have a database, where it will store its lists in **.ser files**. The name of each file is the identifier of the list.

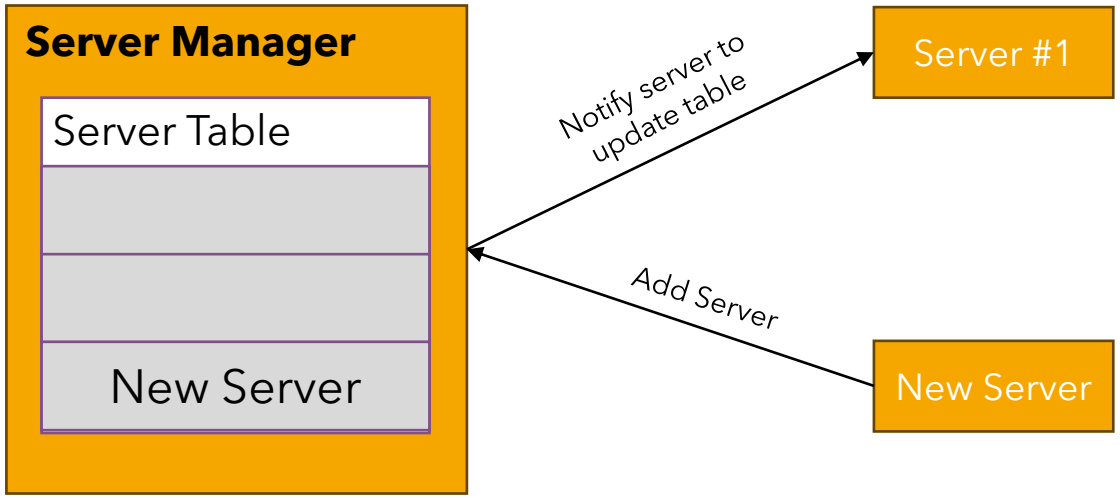
In addition, each server will have an **authentication token** (a .txt file containing a series of numbers). Although security was not our primary concern while developing this project, we felt it was critical to have **some level of protection**.

- When a server attempts to join the cloud, it sends its token to the Server Manager. The manager, who also has a token, then determines whether the received token is valid (in this case, whether the two tokens match). If it is, the new server is added to the server table by the Server Manager.



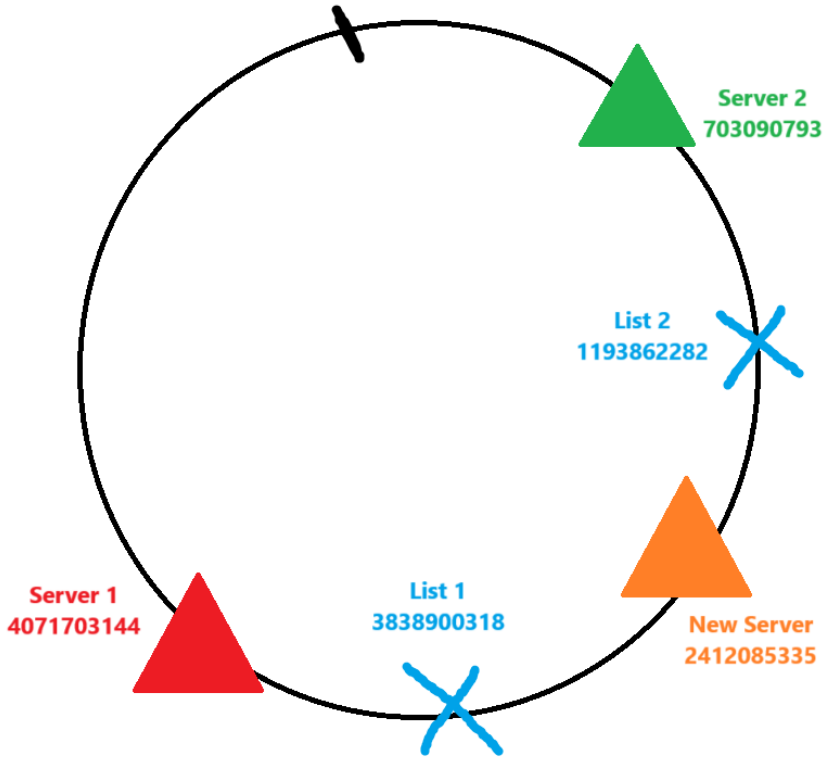
Technical Solution
Server

Each server will also have their **own Server Table**, which is a **copy of the one present in the Server Manager**. This allows for all the servers to know what is going on in the cloud. However, one added task with the replica of this table, is to **ensure the data consistency** between all the servers. Whenever a **new server joins the cloud**, we implemented a solution that lets the **manager notify all the other servers**, alerting them of the new addition. When a server receives this notification, it will **update its server table**.



Technical Solution
Server

The presence of the server table, in each server, is also essential to determine **which lists should be stored in a new server**. When we add another server, the ranges of each existing server node, in the hash ring, will most probably **change**. Therefore, after updating its table, we considered that the server needs to iterate through the set of the list IDs it's storing, check if the ID falls under the range of the new server, and then send the list to the corresponding server (and delete it from its database, if needed).



Example:

Range of Server 1 (4071703144): ~~] 703090793, 4071703144]~~

] 2412085335, 4071703144]

List 2 left the range

Technical Solution
Server

Finally, it's important to highlight the two main features that are included in a server:

- **Pushing a list** – The server receives, from the Server Manager, **the list a certain client wants to push**. In this case, the action of pushing consists of **storing the new version of the list in its database**. More specifically, the server will **retrieve the old list from the database, merge the two lists, delete the original file and write to a file, with the same name, the resulted list from the merging.** In the cases where the user pushes a new list and its ID is already taken remotely, then the Server Manager adds 1 to the ID of the list, until the ID is not taken, and the ID of the list is updated locally, avoiding conflicts.
- **Pulling a list** – The server will receive, from the Server Manager, **the ID of the list a user wants to pull**. If it finds the list in its database, then the server sends the list to the manager, which then can send it to the specific client. If the list doesn't exit, the server sends an error to the Server Manager.

Technical Solution
Other (Message Protocol)

We decided to use **Socket Channels** to allow the efficient and real-time communication and exchange of data across different systems, between the clients, the Server Manager, and the cloud servers. However, in our implementation, we decided to also introduce a **messaging protocol**. With this protocol, the data exchanged between the layers of our architecture will be a **Message**. Each object from this class will be characterized by its **content** and **type**.

On one hand, the **content** can be any Serializable Java object, like an **ArrayList, String, Long**, etc.

On the other hand, the **message type** will characterize the message. We defined many message types, like:

- **Authentication Types** – AUTH, AUTH_OK, AUTH_FAIL
- **List Types** – DELETE_LIST, LIST_DELETED, PUSH_LIST, LIST_PUSHED, PULL_LIST, LIST_PULLED, LIST_NOT_FOUND
- **Server Types** – UPDATE_TABLE, ADD_SERVER, SERVER_NOT_FOUND

Technical Solution
Other (CRDTs)

As it was mentioned previously, two of the key features we need to ensure with our application is high availability and content persistence. For example, when two clients edit the same list concurrently and push each of their changes to the server, our system should be able to deal with the data conflicts.

To handle this, we decided to implement a **CRDT approach**, that ensures that the operations made over the lists are conflict-free and replicated consistently across nodes. In our case, we preferred to use **delta CRDTs instead of state-based CRDTs**, due to the first one achieving more efficiency in terms of the data communicated between replicas.

Technical Solution
Other (CRDTs)

To represent the quantity of each product we use the **CCounter CRDT**.

One of the key components of this CRDT is its **DotKernel**, responsible for managing the state and the handling of the merge operations. This auxiliar CRDT manages a set of "**dots**", each representing a pair composed by a unique identifier and a “counter”. DotKernel handles the addition, removal and merging of these "dots".

The increment method adds a new "dot" with an increased value, while decrement does the opposite. The reset method removes all current "dots". The join method merges the current counter with another counter by joining their dot kernels.

Finally, we also introduced a **readValue method** that computes and returns the current value of the counter by summing up all values in the dotKernel.dotMap.

Technical Solution
Other (CRDTs)

To represent the shopping list, we created a new **AWORMap CRDT** (Map<K, AWORSet<K, V>>). Each key in the AWORMap (which is a String representing an item in the shopping list) is associated with an AWORSet. This set stores the CCounter states associated with that key. CCounter is used to manage the quantities of each item in the shopping list.

We used the AWORSet CRDT since it **resolves conflicts in a distributed environment**. Thus, when there are competing operations (such as additions and removals of items from the shopping list), this set guarantees that additions ("Add-Wins") prevail over removals in case of conflict. **This means that if an item is added and removed simultaneously by different users**, the final state will reflect the **addition of the item.**

Technical Solution
Other (CRDTs)

When adding some quantity of an item from the shopping list, if the CCounter for the specified item **already exists (that is, the item is already in the list),** the function **increments the quantity in the CCounter.**

If **the item does not exist**, **a new CCounter for that item will be created and initialized with the given quantity.** The new CCounter is then added to the AWORMap associated with the corresponding key (the item name). The AWORMap (and by extension the associated AWORSet) updates its state to include this new item and its quantity.

Solution Evaluation

When analyzing our application, there are some limitations that can be considered:

- The Server Manager can be pinpointed as **single point of failure**. Although our solution is designed to be local-first, all the features regarding the cloud **would stop working** if the Server Manager **fails**.
- The **determination of the range that a server is responsible for is made by the hashing function previously described.** Although this hashing algorithm produces distributed/spaced outputs, there could be better solutions for this problem. We could **equally partition** the available ring space, **considering the number of working servers.**
- **Too many messages** are sent when a list is **pulled from the cloud**. Although the client only uses the list received in the first response, all the servers that have the list send it.
- The merging of the CRDTs isn't working as expected.

Video Demo

https://youtu.be/cEB8D7j3JXE