Large Scale Distributed Systems

Shopping Lists on the Cloud - Main Design Challenges and Choices

Grupo 85

André Morais - up202005303
André Soares - up202004161
Aníbal Ferreira - up202005429

Index

In this presentation, we will address several topics regarding our project, that include the **main design choices** and **challenges**. This presentation acts as a final report and includes the following points:

1. Introduction
2. Features
3. CRDT Implementation
4. Node Distribution - Load Balancer
5. Node Replication
6. Handling a Node Failure
7. Conclusions
8. Demo

Introduction

This project explores the creation of a **local-first shopping list application**. The application runs in the user device persisting data **locally**, but also has a **cloud component** that allows users to **share data** among users and offer **backup storage**.

Users can **create** and modify new **shopping lists** via the **user interface**. After creation and until a **list** is deleted, it **exists under a unique ID** (or key) that can be **shared** with **other users**. Users who know that key can **add** and **delete** items to the list, as well as delete it completely. Each **item** is associated with a target **quantity** that represents the amount of items the user **wants to buy** and that can be **increased** or **decrease** accordingly. Since users can **concurrently change the list** and we aim for high availability, we took advantage of **Conflict-free Replicated Data Types (CRDTs)**.

Features

The main features provenient of the design chosen for our application are the following:

- Our system allows the **creation of multiple user accounts**, in the same device.
- Account creation requires cloud connection, but the **login** doesn't, being **local-first**.
- Users can add their **existing account** to a **new device** with exactly the same information.
- Users may login even if the server is down, assuming their account is already stored in the cloud.
- Users can also **delete** their account from the application if they chose to.
- Users can **create, edit or delete** their own **lists,** completely locally.
- Users may also **add**, **modify** the quantity and **remove** items from lists as they please, without cloud connection.
- If not previously shared to the cloud, lists aren't backed up and cannot be accessed by other users.
- If the user decides so, lists **can be shared to the cloud,** by pressing a 'sync' button.
- Sharing to the cloud, stores the list in servers and generates a **unique key** identifier that can be **shared** with other **users**.
- Concurrent list operations may happen, since all **users** with access to a list **can perform different operations** to it, and modify the same items.
- The **conflicts** that can happen in the sync of those operations **are** correctly **handled** to avoid inconsistencies.

CRDT Implementation

Our design allows users to **make changes** to lists **without** requiring a **cloud connection**. The **problem arises** when those changes need to be **passed to the cloud**. This means **conflicts** will arise if more than one person is **changing** any given list at the **same time**.

To combat this, we implemented a **CRDT** so that when users make their **changes** locally and **send them** to the cloud or **receive** other changes via a press of the 'Sync' button, they are **merged correctly**. Those changes include **addition**, **removal**, **deletion** of **lists**, **items** or **users**.

The CRDT is needed in **two scenarios**, when the changes to a given list from the clients are **synced with the server**, and when a server is **replicating** its changes with **other servers** that might have changes of their own as well.

For our implementation, we used the notion of a **PN Counter** and tried to **adapt it** to best suit our needs.

CRDT Implementation

The **PN Counter** is implemented in a way that when a user shares a list, **updates** are **only recorded once they choose to share it**. The initial sharing marks the initial state, and subsequent updates are tracked and synched to the server through a ‘sync’ button press.

Upon adding a shared list to their profile, users receive the latest state. The **‘sync’** button facilitates **sending** and **retrieving updates**, **allowing** users to make **local changes** to multiple **lists** and **items**, **without being connected** to the server.

Each user's updates are locally recorded in a database, with **two tables** for **each item** in a list. The **database** stores **positive** and **negative** changes, enhancing the basic PN Counter concept, and the **quantity** of a given **item** is also stored.

For instance, if a user **adds 2** potatoes and **removes 4** breads from a shopping list, the **dictionary reflects** a **+2** for potatoes and **-4** for breads in the respective rows of the database for that list.

CRDT Implementation

The server merges its CRDT with the user's, using a **max function** and calculates the **new quantity** based on the **changes** in the dictionaries, like in a traditional **PN Counter**. In case of **conflicting updates**, we developed the following function to **determine the final result** of the changes, so that all the databases involved end up with the same values. This ensures **consistency**, preventing conflicts when users receive updates.

```
def quantityChange(self,other):
    change = {}
    for key in other.inc.keys():
        change[key] = abs(self.inc.get(key) - other.inc.get(key)) - abs(self.dec.get(key) - other.dec.get(key))
    return change
```

It does the following: AbsoluteValue(MergedPositive - Positive) - AbsoluteValue(MergedNegative - Negative)

For instance, if two clients modify the **same item**, in the same list between syncs, where the **initial state** is **5 potatoes**. If the first client **adds 2** potatoes while the second one **removes 3** potatoes, the server reconciles changes by calculating this difference, **abs(2 - 0) - abs(3 - 3)**, which will add the quantity of 2 for the second client, ensuring accurate quantity adjustments and **leading to** the same quantity of **4 for both clients**.

Node Distribution - Load Balancer

When a new list is shared, the **load balancer chooses** one of the **servers available** with the lowest load, **to store** the list and respective items. The load balancer plays a crucial role in distributing incoming requests, across multiple servers to ensure **availability** and **reliability**, as well as **resource management**.

The load balancer keeps track of the available servers and their loads, and if a server is not available the updates are redirected to next available server in the list. In case, a load balancer fails, we have **multiple load balancers**, to ensure that the next one is responsible for the distribution.

Each server maintains a local database, and, in the replication phase, **changes** are **propagated** to the **other servers** to maintain a consistent state across the distributed system whilst avoiding a single point of failure.

Node Replication

The node replication is used to maintain a **constant state** between servers. Our choice was to make the system so that each server tries to have the same state as the others, with **changes** being **propagated** among them by exchanging updates through **HTTP requests**.

To propagate changes, the server checks for **updates periodically** and concurrently through **threads**, it requests updates coming from other servers every 5 minutes. For demonstration purposes, we used a smaller time in the video but the 5 minutes mark is a good midpoint for fast updates whilst avoiding a high server load.

Another important aspect to mention is the **handling** of **conflicts** when two or more servers have conflicting changes and attempt to propagate these to each other. This is **solved** once again **by the** employment of our **CRDT**, which merges the changes from both servers to guarantee a **consistent state**.

It should be noted, that in order to **handle temporary failures**, each server has a list of other servers to check for updates, this way there **isn't** a **single** server **responsible** to **pass an update** to a particular server. The **updates** are replicated and **distributed evenly**.

Handling a Node Failure

Our architecture is designed to ensure continuous operation and data consistency, even in the event of a node failure. This is **achieved** through **node replication**, where **servers maintain** the **same information** across multiple nodes, **preventing data loss** if one of them fails. Each node is aware of its **availability state**, contributing to a consistent system operation.

The **load balancer** plays a crucial role in addressing node failures. Before connecting a client to a server, it **verifies** the **server's** online **status**, ensuring that **only available servers handle client requests**. Additionally, servers continuously **monitor each other's** states to determine **availability**, before initiating updates.

To enhance system **robustness**, we implemented **multiple load balancers**. **If one** load balancer **fails**, **others** can seamlessly **take over**, preventing any disruptions. Also, **clients** are **not tied** to a **specific server**, in case of a server failure, the load balancer **redistributes** the **load** to **other available** servers, maintaining a **reliable** and **responsive** system.

Conclusions

In the **complex task** that is designing a distributed system, some **trade-offs** are often **required**. Our design allows for **consistency** between servers and **handling of failures**, may it be in nodes or load balancers. However, in spite of our efforts to minimize loss of data, as mentioned before, our **system is not perfect** and has **weak points acknowledged** by us, but whose solutions would bring drawbacks we can't afford.

Some of the weak points of our system are the following:

- If a server **crashes after receiving** an **update** from a user **and before transmitting** it to any other server, that **data will be lost** and the user will need to make those changes again. We allow this due to the **low probability** of this happening, as well as the **low impact** of that loss data, since the following changes would be handled by another of the available nodes.

Conclusions

Other weak points of our system are the following:

- **All servers maintain** the **same state**, and although this is useful for our system, guaranteeing **strong consistency**, it also means a **high resource allocation** for each server, which is **not ideal** for the **scalability** of a distributed system. A **solution** for this would be **partitioning**, which we chose not to implement as it would generate other **complications**, but that would **improve scalability**.

- As our system stands currently, **each server** has a **list** containing the **existing servers**, three for demonstration purposes, and **updates** are **requested** and **shared** between **all servers**. On a large scale, our solution would still work but these lists **would need** to be **adapted** so that they only store a few nodes of the load balanced group, and **not all** the existing servers. Although **each server communicates** with **more** than **one node**, if the **number of nodes is too high**, the **efficiency** would **drop** and the **scalability** would be **compromised**.

Demo

Now, we will present a short demo that highlights the features of our project: