Local-first Shopping List Application using CRDTs

António Santos
up202008004@up.pt
Faculty of Engineering of University of Porto
Porto, Portugal

José Castro
up202006963@up.pt
Faculty of Engineering of University of Porto
Porto, Portugal

Pedro Nunes
up202004714@up.pt
Faculty of Engineering of University of Porto
Porto, Portugal

Pedro Silva
up202004985@up.pt
Faculty of Engineering of University of Porto
Porto, Portugal

1 INTRODUCTION

The aim of this project is to build a local-first shopping list application that provides a local client-side database that connects to a cloud service used to share data among users and act as backup storage. To this end, this report will go over the architecture to implement and the different services to set up.

Furthermore, with the intent of bringing about high availability and concurrency handling, the usage of Conflict-free Replicated Data Types (CRDTs) was suggested. The types of CRDTs to cover in this project will also be presented in this document.

With regards to the cloud's architecture, it will be built with a large user base in mind, making balancing the stress on the cloud side services a must. Give the scale, this concern must also be extended to data storing, we must therefore keep in mind the need for high availability and durability of information. These topics and our proposed solutions are discussed in more detail below.

2 CRDT

A crucial decision for this project is the designed model of the shopping list data and its components. The basic capabilities of this application are simply item addition and subtraction. To this end, we chose to create two data structures: the *shopping list* and the *product*. The following depicts the defined structure of the data.

2.1 Data Structures

```
{
    id: <string>,
    products: [
        {
            name: <string>,
            quantity: <number>,
            timestamp: <string>
        },
        ...
    ],
    timestamp: <string>
}
```

The *shopping list* data structure will merely serve as a identifiable object congregating a list of *products*. Their composition is as follows:

- *id* - an unique identifier;
- *products* - a list of products, which are themselves CRDTs;
- *timestamp* - a timestamp of the time of the last change made to the list;

Seeing as the products are simultaneously the main object of the shopping list and susceptible to concurrent changes by the end users, it is paramount that their information is consistent. They are defined as follows:

- *name* - a succinct description of the product (e.g. "bananas"), also identifies the item;
- *quantity* - the amount of that product currently to be bought;
- *timestamp* - a timestamp of the time of the last change made to the product;

2.2 Functions

Defining the structure is not enough for establishing a CRDT - it is equally necessary to establish a function that ensures conflict-free data. As such, our design implements the following:

- G-Set (Grow Only Set) - a state based approach of this CRDT allows us to keep track of the products present in the shopping list. Their implementation lies in the *products* attribute;
- PN-Counter (Positive Negative Counter)[2] - PN-Counters' strength lies in their ability to handle increment and decrement operations, something paramount to our approach to the problem; this is achieved through an operation based implementation and by the *quantity* and *timestamp* attributes of the products;

2.3 Other Possibilities

Alongside targeting these basic goals, the application can be expanded to, for example, also let users delete items; this could be achieved by simply adding a *tombstone* attribute to the products or a new *deleted* list to the shopping list, alongside an implementation of a Last-Writer-Wins Element Set CRDT, or a more sophisticated approach using an Add-Wins Set CRDT.

3 ARCHITECTURE

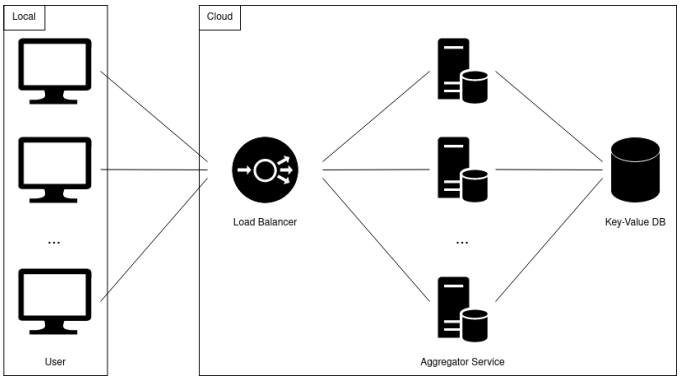

Figure 1: Architectural diagram

The above figure [1] illustrates the proposed architecture of the application at a high level, and each of its components are described in further details in the following sections.

3.1 Local

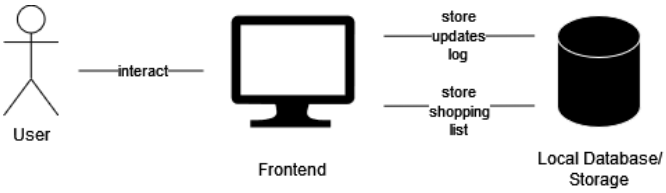

Figure 2: Local diagram

3.1.1 ***Frontend.*** Since the focus of this assignment is on the backend and data storage, our frontend will be a simple and restrict itself to only the most necessary functionalities. In order to allow for easier and more accessible testing and development we will first develop the local frontend as a simple CLI tool with minimal quality of life functionalities and set our sights instead on developing a robust application that fulfills the goals for the user experience. In the future, we intend to build a simple web-app using service workers, as it is a modern paradigm commonly used for current local-first development and carries the advantage of allowing easy deployment, since in a real life environment users that access the application's online web page would be able to access it and use it offline with no other setup required.

3.1.2 ***Database.*** In order to keep local development flowing correctly we must store information locally before distributing it, meaning that we'll need to keep both the state of the shopping lists and a record of the changes done by the user. The latter is simply done by logging changes onto a local logging file, whilst the the former can be accomplished by using a simple and lightweight key-value storage, such as those already built into browsers, or just files.

3.1.3 ***Backend.*** Our backend will execute operations on the local client-side database while also handling remote communication with the cloud service to ensure correct consistency of the stored data. While offline, it will be responsible for storing local changes and, when online, upload these to the cloud. Merge conflicts will not be dealt with by this service, and its only job is retrieving information from the server.

3.2 Cloud

Figure 3: Cloud diagram

3.2.1 ***Load Balancer.*** The first actor of the cloud side of the architecture is the Load Balancer, whose function is implied by its name. The Load Balancer acts as a "middle man" between the incoming traffic and the servers, roughly distributing the load of the requests equally amongst the multiple "Aggregators" and ensuring that the users' response time is more accurate. Its implementation will at first be attempted by the group, seeing as this is a fairly well documented domain which presents an interesting challenge for the team; however, if we are unable to execute it due to time constraints we'll resort to a pre-existing solution. This is most likely to be the facet of the architecture that would fall behind in a real world context, seeing as the geographical component of the traffic is not being taken into account.

3.2.2 ***"Aggregator" Service.*** After the load balancer, each user's request goes through an "aggregator" service. This step is the backbone of the project, as it is where the chosen CRDT's policies are enforced, which means dealing with potential conflicts across multiple users. The service will first decide whether or not the incoming changes result in conflicting outcomes, and resolve those outcomes if necessary.

3.2.3 ***Database.*** The main database of the application resembles the model proposed by the Dynamo[1] team, following a ring like approach which ensures replication of the information through neighboring nodes of a distributed storage system.

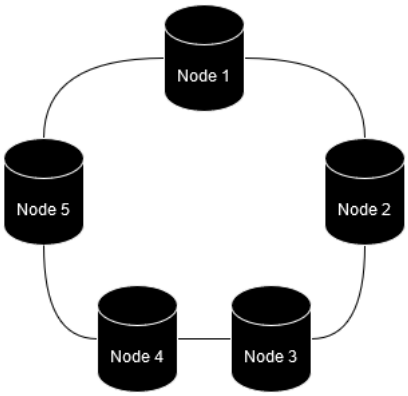

Figure 4: Server database diagram

As figure 4 illustrates, nodes represent parts of the bigger database, this allows us to easily replicate data whilst simultaneously ensuring its' availability in cases of failures, this is done by partitioning the keys into segments and dividing their spaces amongst a number of servers, the previous example featuring 5.

High availability and durability of information is ensured by having nodes be responsible not only for a predetermined "key space" but also for one of its closest N predecessors, which means Node 1 would be responsible for keys within its designated range, as well as the ones on 5 and 4, if N equals 2. This introduces a deal of complexity that makes adding other features difficult (e.g. increasing the number of nodes without any downtime), resulting in them not being contemplated for inclusion in this project.

4 CONCLUSION

In conclusion, our cloud architecture was carefully considered to support millions of users, as was proposed. Data access bottlenecks are avoided due to the load balancer distributing the requests evenly to avoid hotspots and data sharding in the cloud database, which provides multiple access points. Thus, any user can access and modify a given shopping list with minimal delay and close to no conflict issues. The local architecture is simple, yet effective enough to ensure consistent behaviour when modifying each shopping list and requesting/sending an update to the server.

REFERENCES

[1] G. DeCandia, D. Hastorun, M. J. G. K.-p. A. L. A. P. S. S. P. V., and Vogels, W. Dynamo: Amazon's highly available key-value store.
[2] Goel, A. Pn-counters, 2016.