Local-First Shopping List Application

SDLE - FEUP | MEIC 1º (2023)

João Alves - up202007614

Marco André - up202004891

Ricardo de Matos - up202007962

Introduction

Objective: Adoption of distributed system strategies in order to create a local-first shopping list application.

The application should support the following features:

- Create multiple shopping lists where users can add and check-out items
- Lists can have items with quantities that can change (+/-)
- Share lists via links for list collaboration
- Local-first functionality

Local-First Approach:

- Fully functional offline with seamless experience
- Data stored with local persistence
- Server connections only when necessary
- Synchronizes when connection is available

Architecture:

- High availability, even with consistency trade-offs
- Fault tolerance even with millions of users
- Influenced by Amazon's Dynamo

Technologies

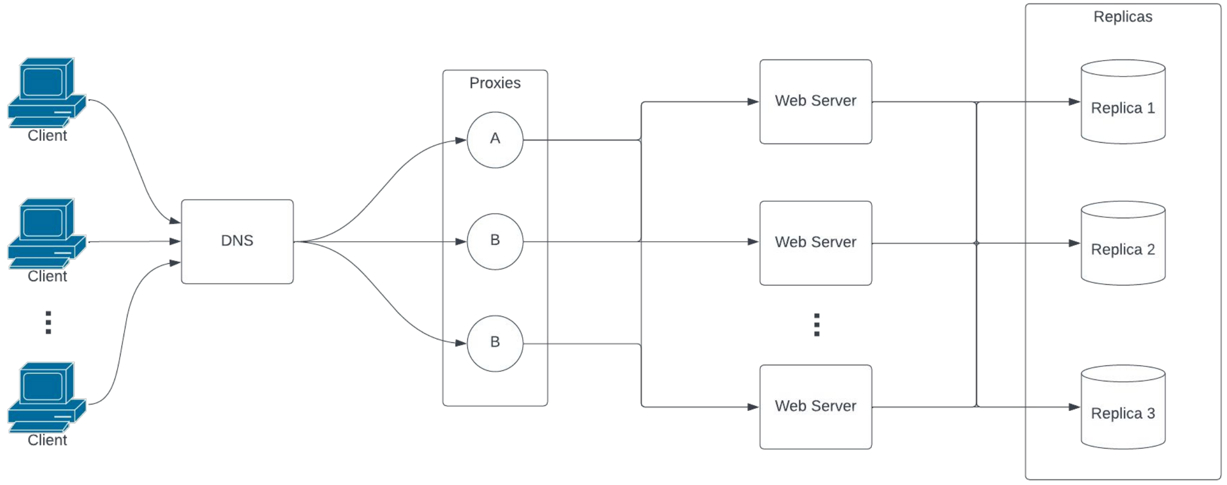
For the frontend and backend we used: **JavaScript** with the following packages:

- **better-sqlite3**: for **database** management
- **bintrees**: **Red-Black** tree used in the **consistent hashing**
- **express:** used to create the endpoints
- **live-server**: to start the **2 frontend UIs**
- other packages for punctual needs such as fs, path, etc

For the database replicas we used: **SQLite**.

Architecture

The architecture of our local-first shopping list application (presented below) is designed to meet the requirements of **high availability**, **data synchronization**, dynamic adaptation to work load, and **fault tolerance** in multiple places of our architecture.



Note: this **does not mean** that every node in the image **is a different server**, rather a different **logical component**

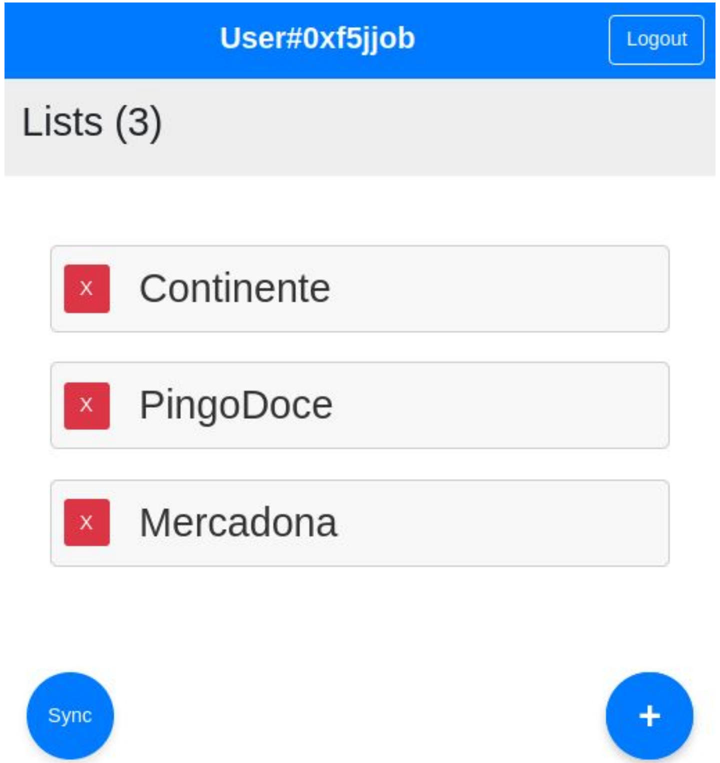
Performed Work

- **Local Data Persistence -** with no need to have internet connection
- **Cloud Component -** provides backup storage for user data and sharing.
- **Shopping List Creation and Management -** adding/removing lists and sharing using **url**
- **Collaborative Editing -** allowing multiple users to edit a list
- **CRDTs -** to merge different/conflicting data and ensure consistency
- **Item Management -** adding/removing items quantities needed and checking when all are acquired
- **High Availability -** designed to support millions of users and to avoid data access bottlenecks
- **Data Independence -** independence between list, which are partitioned into different databases.
- **Load Balancing -** to ensure good use of resources and avoid bottlenecks

Local-First Functionality

The local-first application operates by prioritizing data on the user's device, ensuring swift access and editing capabilities even without a server connection. However, to facilitate data sharing across multiple clients, it synchronizes with cloud storage (which also serves as data backup). Local-first applications aim at:

- Primary copy of the **data** on the **local device**
- Good **offline support**
- Better **privacy** and **security**



Scalability

In an initial phase, we implemented a simple **client-server** setup that could never work at scale with a one-to-one interaction. As we aim *millions of users*, we need a **set of web servers nodes** processing multiple user requests, which is a **horizontally scalable** solution.

To provide even more scalability, a simple **DNS** was developed. This provides a way to distribute the client requests throughout the proxies.

The **proxies** adapt **dynamically** to the changes in the system and allows the balance of requests to the web servers.

The system can be configured to have a large amount of proxies, web servers and replicas according to the needs of the application.

Data Replication and Consistency - Quorum

Data replication is a must-have in order to provide the users with high application availability.

To handle data operations with multiple replicas, while still having a relatively good consistency for the user’s needs, we implemented a sloppy quorum replica consensus, as described in the Amazon Dynamo Paper (where $W + R < N$).

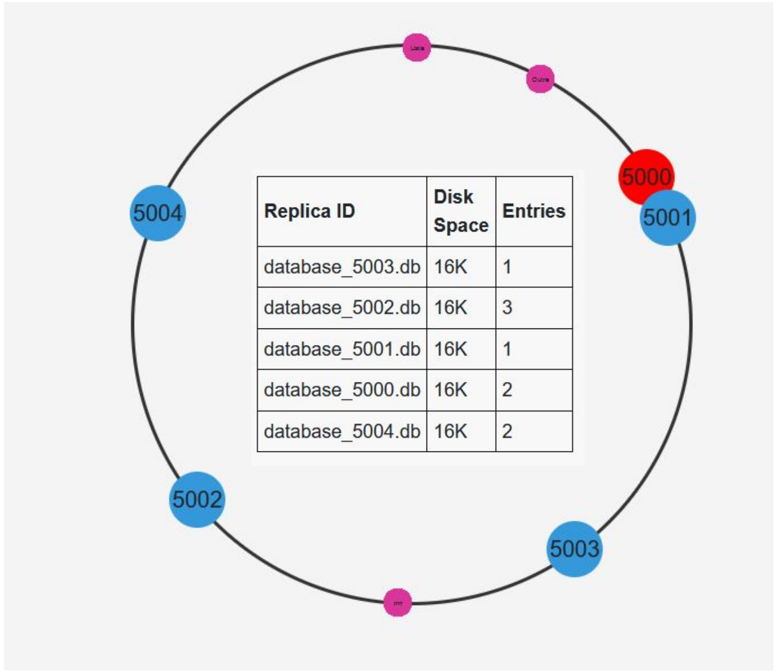
It is a good solution for our application, as it guarantees a more smooth operation while sacrificing some consistency. For example, our default version of the architecture has 5 replicas, $R = 2$ and $W = 2$ as a viable solution to still provide some consistency while being more tolerant to failures.

Our web workers also provide extra consistency to the replicas, as they update the outdated replicas when needed, and tend to a state of consistency among the replicas of the quorum.

Data Partition - Consistent Hashing

Implemented features:

- **Virtual nodes** for better distribution of load (although not used in this case)
- Fault tolerance of nodes in the ring, **Hinted Handoff** - in this example node 5000 is down
- **UI** to monitor, **kill** and **restart nodes** in the ring - as presented to the right
- **Red-Black Tree** to bound loads (O(log n) Search)



Replica ID	Disk Space	Entries
database_5003.db	16K	1
database_5002.db	16K	3
database_5001.db	16K	1
database_5000.db	16K	2
database_5004.db	16K	2

Nodes in blue/red and lists in pink

Data Replication and Consistency - CRDTs

The team implemented **G-Counter**, **PN-Counter** and a custom **Shopping List** CRDT that uses the first two.

The Shopping List CRDT has the following features:

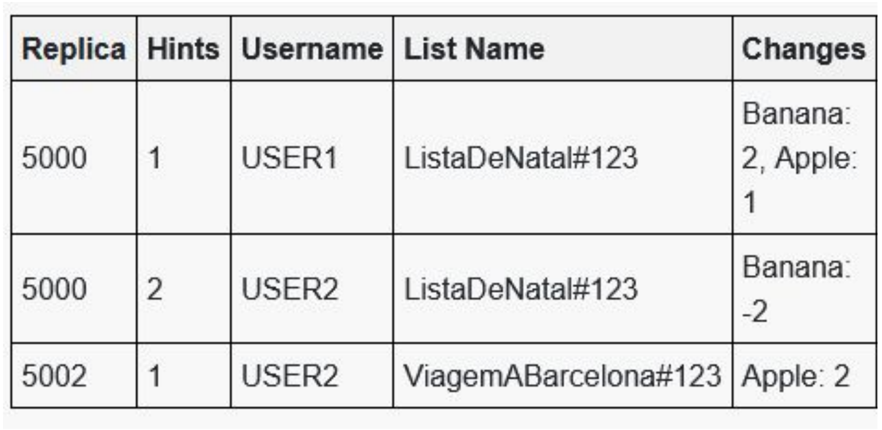
- An **addition/increment** to an item represents **a need for the product**
- A **deletion/decrement** represents the **buying/canceling** of **that need/quantity**
- Users **send their commited changes/deltas** into the cloud storage
- Every change is **tagged** with an unique ID timestamp to **identify the change**
- **Order** of tags is **not important** has every node will arrive to the same conclusion due to **commutativity of sums**
- **Users request** changes after the last read change
- **Garbage collection implemented** to clean older commits by merging them into a single one although not taking into account the last user read change (still to implement)

Hinted Handoff

Hinted handoff serves as mechanism for coping with replica failure by storing the data and other relevant metadata that was meant to the failing replica and pushing this informations to the intended recipient once the replica is detected as operational again by periodical heartbeat checks.

This strategy provides resilience when a replica recovers from failure, allowing it to catch up to the remaining replicas and participate on the consensus.

As seen on the control panel UI to the right, the data and associated meta information needed to later return to the failing replica are all stored.



Replica	Hints	Username	List Name	Changes
5000	1	USER1	ListaDeNatal#123	Banana: 2, Apple: 1
5000	2	USER2	ListaDeNatal#123	Banana: -2
5002	1	USER2	ViagemABarcelona#123	Apple: 2

Load Balancing

Load balancing plays a crucial role in our **scalable architecture**, ensuring efficient **user request distribution** and **system availability**, even during network stress.

We implemented a strategy using **DNS** and **proxies** as **intermediaries to evenly distribute incoming traffic** among the available web server nodes. The DNS uses a **Round-Robin algorithm** to select the proxy that should take the requests of the client. The proxies implement a different load balancing strategy. They use a **Least Time** algorithm that sends the requests to the server that has the lowest response time.

This approach prevents web worker overloads, **optimizes resource usage**, and **enhances system performance**, while also contributing to fault tolerance by seamlessly redirecting traffic in case of failures.

This dynamic and adaptive load balancing mechanism responds to changing workloads and traffic patterns, making it vital for our architecture's scalability and high availability.

Fault Tolerance

Embracing the fact that the failing of any node in our network architecture is inevitable, the team has built a **fault resilient system** to assure a **good user experience even under network stress.**

The quorum consensus can still be performed even if replicas fail temporarily by **relaxing the quorum** to still allow operations to take place at the expense of momentary consistency. Moreover, the **hinted handoff** mechanism previously described also provides additional resilience when a replica recovers from failure.

It's noteworthy to mention **these solutions address failures of short duration** and long term failures would require more complex architectural solutions.

When a server fails, the solution is pretty straightforward as **horizontal scalability** allows for easy addition of more worker nodes and the proxy can easily **redistribute the load balance**.

Finally, a single proxy to redirect client requests would present a bottleneck and a single point of failure so we developed a **set of actively working proxies** that each receive client requests from a **DNS**, which attributes a user to a proxy on each session login based on **Round-Robin algorithm**.

Conclusion

Throughout this project, we developed a local-first shopping list application that includes code that runs on users' devices, enabling local data persistence, and a cloud component for data sharing and backup storage.

Users can create and share shopping lists via a unique ID and collaborate seamlessly if online due to the implementation of Conflict-free Replicated Data Types (CRDTs).

The cloud-side architecture was heavily inspired by the Amazon Dynamo paper and was carefully designed to achieve high availability while avoiding data access bottlenecks, leveraging independent data sharding for each list via a consistent hashing replica ring.

This project successfully met the proposed goals, delivering a robust and scalable implementation for an efficient user experience while both offline or online.

End