List Llama - Shopping Lists on the Cloud

Large Scale Distributed Systems

Ana Rita Oliveira - 202004155
Jorge Sousa - 202006140
Mariana Teixeira - 201905705
Matilde Silva - 202007928

Masters in Informatics and Computing Engineering

November 2023

Contents

1 Introduction

In this project we will develop an online shopping list following a local-first [5] approach. The application will provide the features below:

- Add and remove shopping lists or its items.
- Mark an item as bought.
- Share shopping list with other users.

Because multiple users will be able to change a shopping list simultaneously, we must have tools to manage the conflicts that might occur. To solve this problem, we will create our implementation of CRDT's [1] (Conflict-free Replicated Data Type).

2 Background

During the development of the application, we will assume that the user will be able to share and communicate a shopping list's URL, outside of the application, and also that the application will be used by millions of users.

3 Architecture

3.1 Local-First

Following the project requirements, we will implement a local-first software, i.e. local data changes are prioritized and client-made alterations will be synced once network connection is established.

3.2 Scalability

Taking into account our Distributed System's theoretical knowledge, we will create a system that employs one or more nodes (servers). Ideally, each server perfectly replicates one another. In other words, data between servers is consistent. This approach ensures availability, in case of node failures.

Additionally, we will use RabbitMQ [2], which can send up to millions of messages per second, but it requires multiple brokers to do so.

3.3 Replication

As mentioned before, the servers will be replicated, which brings about some issues with consistency. To make up for this, the following solution is proposed: whenever a client makes a change to a given shopping list, one of the servers will pickup on this alteration. The receiving server will then ask all other servers for their copies of the shopping list. The receiving server will then merge all copies from servers and client, redirecting the new shopping list to all servers and also the client. The integrity of the list is maintained by the Conflict-free Replicated Data Types (CRDT's).

Replicas of a CRDT object can be updated without synchronization and are guaranteed to converge.

3.4 Consistency

The chosen consistency model is Eventual Consistency [9], which guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

To ensure the consistency of the distributed system and its data, Conflict-free Replicated Data Types (CRDT's), specifically Add-Win-Sets [4], will be used. CRDT's are a data type structure that makes distributed data storage systems and multi-user application simpler and, in this case, Add-Win-Sets is a set data type in which additions take precedence over removals.

CRDT's will be used in both the client side and server side, as well as in the contents of the exchanged messages.

3.5 Data Partitioning and Distribution

In this project, data partitioning and distribution will be achieved through consistent hashing and the employment of virtual nodes.

When the broker receives the client message, it will employ a hashing operation which will yield a number from 1 to M, with M being the number of virtual nodes. Similarly, each virtual node on the hashing ring will be mapped to a physical node. The number that the hashing function returns, indicates which virtual node and, consequently, which physical node will be tasked with dealing with the new changes.

These two strategies ensure even load distribution and consistent storage, guaranteeing no single resource will be overloaded.

3.6 Fault Tolerance

In case of failure of one or more nodes, the messages that would be directed for them will then be redirected to the next available node on the hashing ring. This redirection ensures that the data remains accessible, even when one or more nodes are temporarily unavailable.

The information on the faulty node's local database, when it becomes operational again, will eventually be caught up with the changes that occurred in the meantime, as is guaranteed by the Eventual Consistency model.

4 Implementation

To build a client-side application, we will use React [6] for the frontend that will communicate with a Go [3] backend. Each client will store its local data in a SQLite3 [7] database.

The tool that we will use for the message communication is RabbitMQ, using the AMQP [8] (Advanced Message Queuing Protocol) messaging protocol.

AMQP is a binary protocol and defines strong messaging semantics.

References

[1] Marc Shapiro Carlos Baquero Valter Balegas Sérgio Duarte Annette Bieniusa Marek Zawirski, Nuno Preguiça. An optimized conflict-free replicated set. https://lip6.fr/Marc.Shapiro/papers/RR-8083.pdf, 2012.

[2] AWS. Rabbitmq vs kafka. https://aws.amazon.com/compare/the-difference-between-rabbitmq-and-kafka/, 2023.

[3] Go. https://go.dev/.

[4] Annette Bieniusa Martin Kleppmann and Marc Shapiro. Crdt glossary. https://crdt.tech/glossary#:~:text=Add%2Dwins%20set%20(AWSet)%3A,element%20is%20in%20the%20set., 2023.

[5] Peter van Hardenberg Mark McGranaghan Martin Kleppmann, Adam Wiggins. Local-first software. https://www.inkandswitch.com/local-first/, 2023.

[6] React. https://react.dev/.

[7] Sqlite. https://www.sqlite.org/index.html.

[8] Wikipedia. Advanced message queuing protocol. https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol, 2023.

[9] Wikipedia. Eventual consistency. https://en.wikipedia.org/wiki/Eventual_consistency, 2023.