Shopping List Application Design

Large Scale Distributed Systems

Bárbara Rodrigues
up202007163

Nuno Silva
up202005501

Sofia Costa
up202300565

Tiago Ribeiro
up202007589

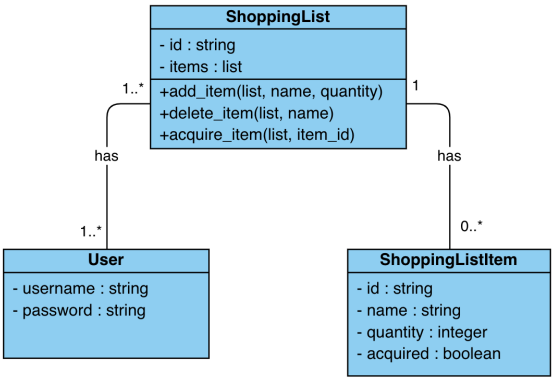
Project Description

In this project, it is intended to create a shopping list application. The application includes code that operates on the user's device, allowing for local data storage, as well as a cloud component for data sharing among users and providing backup storage. Using the user interface, individuals can create new lists, open them and access shared lists of other users, through a password. With a list chosen, the user can edit it, adding and removing items, changing the quantities or marking them as acquired or not.

Design

Conceptual Model

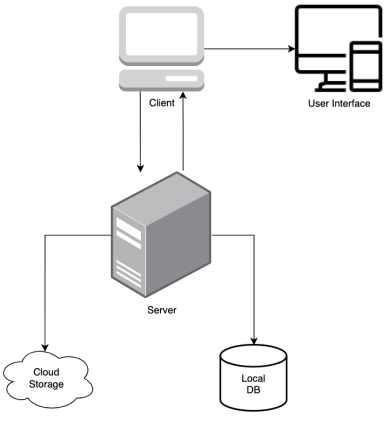
The domain model of this project consists of three entities namely, Shopping List, Shopping List Item and User. A Shopping List is created by a User that is authenticated in the application. The user is allowed to add, delete, and update items of the list. A Shopping List Item is described by its item name, the desired quantity, and a flag that lets the user know if the item has been acquired or not. A User can have multiple Shopping Lists and a Shopping List can be owned by multiple users if shared.



System Design

System Architecture Diagram

The system's architecture can be represented by the following diagram. The application supports multiple servers and multiple clients. The servers are responsible for managing user authentication, list synchronization, and local and cloud storage, which stores user credentials and shopping lists. Each client runs the shopping list application and communicates with the user interface.



In the backend, multiple identical servers are deployed to handle incoming client requests. These servers are equipped with the same application code and configurations to ensure consistency.

A load balancer acts as the traffic cop, intelligently distributing incoming requests among the backend servers. It uses a load-balancing algorithm, namely a weighted round-robin algorithm, to allocate requests evenly across the server pool. The load balancer ensures that no single server becomes a performance bottleneck. By spreading the load, it prevents any server from becoming overwhelmed and guarantees minimal latency, faster response times, and a smooth user experience.

As user traffic increases, we can easily scale our system horizontally by adding more servers. The load balancer seamlessly incorporates these new servers into the rotation, ensuring that we can accommodate more users without service degradation. It can also turn them off if they are not needed.

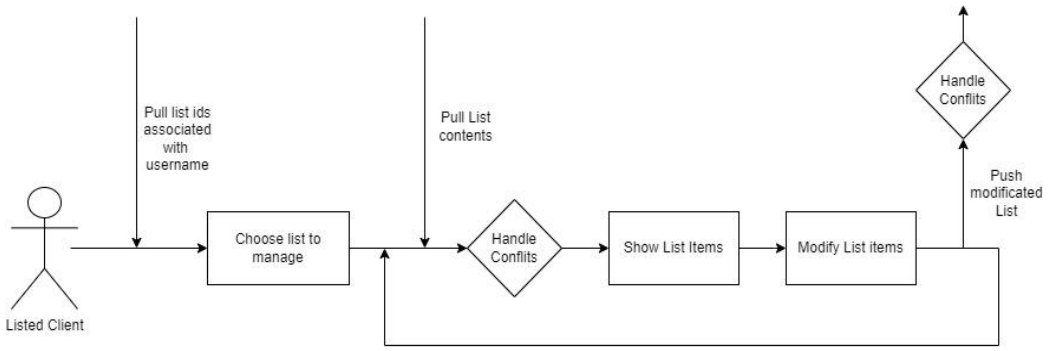
In the event of a server failure, the load balancer immediately reroutes traffic to healthy servers, ensuring uninterrupted service and reducing downtime. This high availability is crucial for maintaining a reliable user experience.

Local-First Approach

Our Shopping List application embodies this principle by enabling users to create, edit, and manage their shopping lists primarily on their local devices, without being heavily dependent on centralized servers.

One of the principles of the Local-first approach is the ability for users to work with their data even when their devices are offline or in a partially connected state. Our application is designed to support offline functionality, allowing users to edit their shopping lists without requiring a continuous server connection. This feature ensures that users can access and manage their lists at any time, whether they are connected to the server application or not.

Our application also adopts the concept of real-time collaboration, enabling multiple users to work on a shared shopping list simultaneously, even when offline. To address the challenge of potential conflicts that may exist when merging data from different users, we have defined a plan to implement Conflict-Free Replicated Data Types (CRDTs). CRDTs allow us to intelligently resolve conflicts and ensure that the shopping lists remain consistent and up to date.



Data synchronization strategy - CRDTs

In order to replicate the data across multiple computers we plan on using two different types of CRDTs to deal with the different elements in a shopping list. The shopping list consists of a list of unique items with information about what they are, the desired quantity and if they have been acquired or not. We plan on using a LWW-Element-Set to deal with the items on the shopping list and a PN-Counter to deal with the counters present in the amount of items that I want to buy.

LWW-Element-Set:

This CRDT consists of two sets: an “add set” and a “remove set”, with a timestamp associated with each element. Elements are added or removed by inserting the element into the add set or to the remove set respectively, with a timestamp. An element is a member of the LWW-Element-Set if it is in the add set, and either not in the remove set, or in the remove set but with an earlier timestamp than the latest timestamp in the add set. Merging two replicas consists of taking the union of the add sets and the union of the remove sets. When timestamps are equal, the "bias" of the LWW-Element-Set comes into play. A LWW-Element-Set can be biased towards adds or removals. LWW-Element-Set allows an element to be reinserted after having been removed.

The methods used in this CRDT are:

- `add` : Add a new element to LWW
- `remove` : Remove element from LWW
- `lookup` : Checks for the presence of an element in LWW
- `compare` : Checks if the given LWW is a subset of another LWW
- `merge` : Merges two LWW and returns a new LWW

PN-Counter:

Consists of two separate counters to represent the positive and negative contributions. This counter supports both increment and decrement operations. It combines two G-Counters namely “P” (for incrementing) and “N” (for decrementing) counters. The value of the counter is the value of the P counter minus the value of the N counter. Merging involves merging the P and N counters independently.

The methods used in this CRDT are:

`increment` : Increment the positive counter for the specified replica

`decrement` : Increment the negative counter for the specified replica

`merge` : Merge the positive and negative counters with another PNCounter

`calculate` : Calculate the final value by subtracting the negative counter from the positive counter