SDLE - Shopping App: Design Plan

ANDRÉ LIMA, GUILHERME ALMEIDA, and MIGUEL MONTES

This paper introduces a local-first [4] shopping list application with cloud synchronization. In response to most applications' reliance on internet connectivity, this system prioritizes offline capabilities for user collaboration. It details the architectural design, enabling offline list management across multiple devices without access control. For data consistency and conflict resolution, an initial approach is outlined, relying on the Last-Writer-Wins strategy and using the local browser clock to prioritize recent modifications. Improvements to this strategy are also outlined, namely relying on Conflict-free Replicated Data Types (CRDTs) [7] to enhance reliability and user experience. By leveraging load balancers and a distributed key-value database inspired by Amazon's Dynamo, the system ensures high availability and scalability.

Additional Key Words and Phrases: Local-first, Shopping List, Cloud Computing, Data Sharing, CRDT, Scalability

1 INTRODUCTION

Personal electronic devices such as smartphones have become an integral part of our daily lives. We use them for various tasks like taking notes, listening to music, and working. However, most of the apps that we use on these devices require an internet connection, which can lead to a poor user experience, depending on external factors such as connection speed and mobile data plans.

Recently, a new type of software has emerged to address this issue. Local-first software is a novel approach that focuses on providing a great offline experience to users. It "enables both collaboration and ownership for users" [4] by first focusing on the offline features of the application and then integrating the online features that are available.

In this paper, we will present our architectural approach to a local-first shopping list application that includes a cloud component for synchronization. With this application, shopping lists can be visualized and modified even when the user is offline. The changes made are synchronized with other users once the user goes back online. We will also discuss the critical aspects of designing such an application, with a focus on reliability, data persistence, and scalability.

2 SYSTEM REQUIREMENTS

In this project, it is expected that we develop an application capable of managing and sharing a shopping list across multiple devices. The requirements are as follows:

- There is no need for access control: each shopping list has a unique URL associated with it and any user who is privy to that URL should be allowed to perform any action upon the corresponding shopping list.
- New shopping lists can be created and existing ones can be deleted.
- Users should be capable of adding or removing items to and from the list.
- Each item in the list should be of one of two types: "single" — where the item can be marked as bought after a user has purchased it —, and "multiple" — where the item has an associated quantity that decreases each time a user buys it.
- Users should be capable of modifying a shopping list at any time, even when there are other users modifying the same list simultaneously or when the user doesn't have an active connection to the application back-end.
- The system should be designed in a way that allows it to handle millions of concurrent users.

Authors' address: André Lima, up202008169@up.pt; Guilherme Almeida, up202006137@up.pt; Miguel Montes, up202007516@up.pt.

3 SYSTEM OVERVIEW AND ARCHITECTURE

In Figure 1 we can see the Architectural Overview of our app.

The system is divided into two major blocks: front-end and back-end.

3.1 Front-end: An offline-first web-page

The front-end will be developed as an offline-first web page. Typically, web pages are only accessible when the user is online. However, using the Service Workers API [6] on modern browsers, we can cache our page's assets, such as HTML, CSS, and JavaScript, on the users' browsers. This makes it possible to offer an offline experience on our web page.

The Service Worker acts as a proxy for the application front-end. It enables background synchronization and manages offline operations. Additionally, it has its own local IndexedDB, which is an in-browser key-value database [5]. The local IndexedDB ensures that users can make modifications, add or remove items from their shopping lists, and perform other operations even without an active internet connection.

Thus, this setup allows the application to function seamlessly, even in offline scenarios, by persisting data locally. Therefore, we can achieve a local-first design that provides a better experience for our users.

To ensure the uniqueness of the list identifiers, we have opted to use a Universally Unique Identifier (UUID). "While the probability that a UUID will be duplicated is not zero, it is generally considered close enough to zero to be negligible" [9], which is enough for our system to be considered reliable.

3.2 Backend: Managing Data and Synchronization

3.2.1 Load Balancers

Since we are developing our app to be capable of handling millions of daily active users, we need to design the back-end with high scalability in mind. To achieve this, a series of techniques will be used to distribute the system load across several servers.

For this scenario, we have found that a tiered approach works best. The core of the approach is a DNS load balancer [2]. This load balancer is capable of distributing traffic across regional load balancers, which then distribute incoming requests among various instances of our application's synchronization service, described in section 3.2.2, based on the current load of each instance. This

approach results in better latency, as servers closer to the users respond to their requests, and higher throughput as well as better availability, as servers are less overloaded with requests.

We are exploring another option for our synchronization service, which involves the use of serverless computation. This approach is easier to set up and deploy, and explicit load balancing would no longer be needed since that is taken care of by the hosting provider automatically. The service's image can be easily deployed across multiple data centers worldwide, which will allow us to achieve lower latency overall. Serverless functions have the ability to easily scale horizontally and, therefore, create more instances of our service as traffic increases, and destroy instances as traffic decreases, ensuring that the system can handle varying amounts of concurrent users.

3.2.2 Synchronization Service

The synchronization service is responsible for ensuring that the state of each shopping list is as up-to-date as possible. To achieve this, every time a user goes online and the service worker makes a synchronization request, the synchronization service will receive that request and merge the user's version of a given shopping list with the central version of that list. The central version of the shopping list will be stored in the database described in the next section and, as such, the synchronization service will need to connect to it via HTTP, for instance.

Furthermore, the synchronization service will also be responsible for semantically merging the different versions of a key on the database, in the event that a concurrent write occurs.

3.2.3 Database

The synchronization service layer connects to a distributed key-value database system. The design of this database is heavily inspired by Amazon's Dynamo database system [3]. In essence, this layer is comprised of various nodes, each consisting of a local database and its respective controller. The controllers are responsible for replicating the data and routing read and write operations across the database nodes efficiently. They communicate and synchronize with each other to maintain the high availability and eventual consistency of the data, even in the presence of network partitions or machine faults.

Consistent hashing is used by Dynamo to partition data across nodes, ensuring that all reads and writes for a specific key will be directed to the same set of nodes. For that reason, we will utilize the shopping list identifier as the key in our database system to store all relevant information regarding that shopping list. This not only means that we can expect a reasonable level of data consistency but also that the database load will be evenly distributed across all nodes since shopping list identifiers will be randomly generated and nodes are assigned a random portion of the hash space.

3.3 Consistency Strategies: Last-Writer-Wins and CRDTs

The system will employ different consistency strategies to manage data conflicts and ensure coherence within the system in two different phases. This involves reconciling differences between distributed data copies, by "exchanging versions or updates of data between servers (also known as anti-entropy); and choosing an appropriate final state when concurrent updates have occurred, called reconciliation" [8].

3.3.1 Phase 1: Last-Writer-Wins

Initially, our system will utilize the Last-Writer-Wins strategy, which is a widespread approach to reconciling differences between multiple versions of data, prioritizing the most recent user modifications to resolve conflicts. By using this approach, our criteria for maintaining data consistency relies essentially on ensuring that recent changes take precedence over older ones.

To determine who is the last writer, our system will rely on the use of the local browser clock, when the synchronization request is made. This approach ensures that the system prioritizes the version of a shopping list that was more recently synchronized, allowing the system to resolve conflicts across different devices. Although this method is very naive, it will allow us to iterate quickly and eventually improve to a more refined solution, such as CRDTs.

3.3.2 Phase 2: Conflict-free Replicated Data Types

In a future phase, we aim to smoothly transition the system towards the integration of Conflict-free Replicated Data Types [7]. This transition is designed to establish a conflict resolution mechanism among distributed versions of a shopping list, without the need for extensive coordination or synchronization procedures typically associated with traditional distributed systems. CRDTs facilitate the convergence of all replicas to a consistent state, even in edge-case scenarios involving concurrent modifications. This integration is anticipated to not only enhance the system's reliability and enable strong eventual consistency [8], but also to improve user experience.

Our implementation will use an Enable-Wins Flag CRDT for items of type "single". This flag will keep track of whether an item has been bought or not. In case of concurrent modification, the flag will favor the enabled state, which indicates that the item has already been bought. This behavior is desirable as it prevents users from mistakenly buying an item that has already been bought.

For items of type "multiple", we plan to use a pair of CCounter CRDTs [1]. This type of CRDT can handle increments and decrements, which allows us to track the number of items that need to be bought and the number of items that have already been bought. We also chose this CRDT because it can be embedded within an ORMap CRDT [1].

To group these counters into a single data structure, we decided to use an ORMap. The keys on the map will be calculated based on the name of the item and the type of the item (single or multiple). Having this setup, we can effectively differentiate the cases where we use an AWFlag from the ones where we use a CCounter.

REFERENCES

[1] Baquero, C. Reference implementations of state-based crdts that offer deltas for all mutations. https://github.com/CBaquero/delta-enabled-crdts, 2018. [Online, accessed 8-November-2023].

[2] Cloudflare. What is DNS-based load balancing? https://www.cloudflare.com/learning/performance/what-is-dns-load-balancing/, 2023. [Online; accessed 2-November-2023].

[3] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *ACM Symposium on Operating System Principles* (2007).

[4] Kleppmann, M., Wiggins, A., van Hardenberg, P., and McGranaghan, M. Local-first software: You own your data, in spite of the cloud. https://www.inkandswitch.com/local-first/, 2019.

[5] MDN Contributors. IndexedDB API — MDN, Mozilla Documentation Network. https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API, 2023. [Online; accessed 2-November-2023].

[6] MDN Contributors. Using Service Workers — MDN, Mozilla Documentation Network. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API/Using_Service_Workers, 2023. [Online; accessed 3-November-2023].

[7] Wikipedia contributors. Conflict-free replicated data type — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Conflict-free_replicated_data_type&oldid=1181361639, 2023. [Online; accessed 2-November-2023].

[8] Wikipedia contributors. Eventual consistency — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Eventual_consistency&oldid=1164025873, 2023. [Online; accessed 3-November-2023].

[9] Wikipedia contributors. Universally unique identifier — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Universally_unique_identifier&oldid=1184109051, 2023. [Online; accessed 8-November-2023].

A APPENDIX

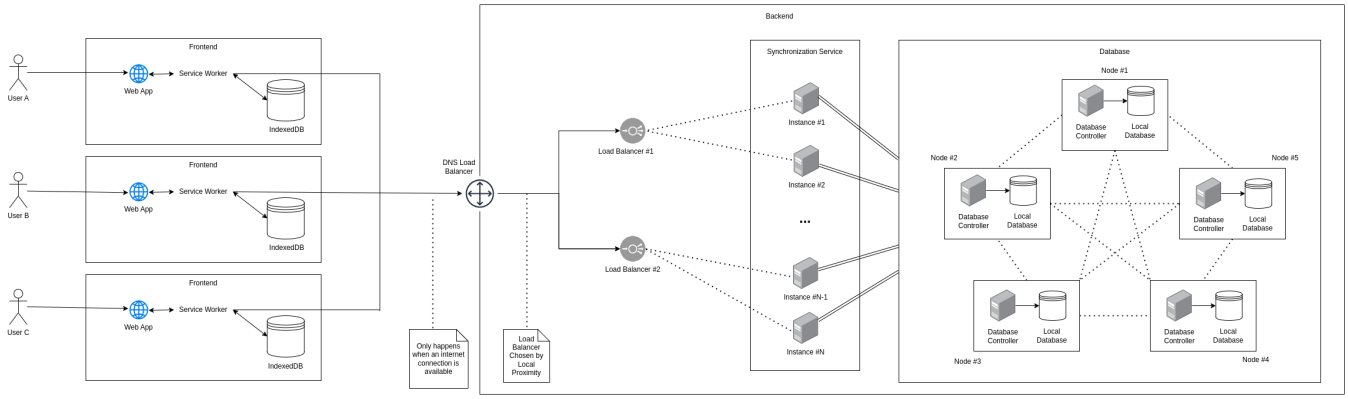

Fig. 1. Architectural overview of our system.

SDLE - Shopping App: Design Plan