Shopping List on the Cloud

André Costa - up201905916
Bárbara Carvalho - up202004695
Fábio Sá - up202007658
Luís Cabral - up202006464

Class: T05

Course: Large Scale Distributed Systems

Master's Degree in Computer Engineering and Computation

Teacher: Carlos Baquero-Moreno

1 TECHNOLOGY

Selected client-side technologies prioritize simplicity and user-friendliness, emphasizing a seamless experience, particularly in web applications where ease of installation and use is crucial. The chosen technologies are:

- *Node.js*, for client and server side applications;
- *SQLite3*, for database management system;

Furthermore, selected technologies and libraries will be employed for the implementation of distributed system connections, cloud infrastructure management, and the maintenance of integrity and consistency:

- *ZeroMQ*, for high-performance asynchronous messaging;
- *UUID*, for the generation of unique identifiers across the entire system;
- *Async Mutex*, to inhibit issues of concurrency between threads;

2 LOCAL FIRST

The primary focus initially is to attain a *Local First* [5] behavior. To achieve this, persisting data from recognized lists is crucial. In the initial phase, the client app checks for the existence of a local database:

- If present, loads its content (lists and items);
- If absent, creates an empty database following the predefined schema presented in [Figure 1]

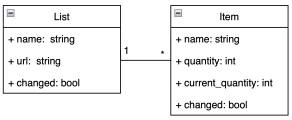

Figure 1: Database schema

Note that the boolean attribute 'changed' is crucial for identifying, in adverse conditions, which lists or items have been modified by the client but are not yet in the cloud backup.

To enable the sharing of shopping lists between users, two requirements must be met simultaneously when they are created:

- The list must be instantiated locally, following the Local First approach;
- The URL must be unique throughout the system and serve as the identifier for that specific list;

If the URL construction relies on the list name and/or creation timestamp, conflicts may arise in the system. To address this concern, a potential implementation is based on *UUIDs* [8]. UUIDs (*Universally Unique Identifiers*) are globally unique identifiers that ensure uniqueness throughout the system. In our case, we will opt to use version 4 of UUIDs, which provide a high probability of uniqueness as they are based on random data. This makes them suitable for generating unique URLs in a distributed system where nodes cannot communicate initially.

Author's address:

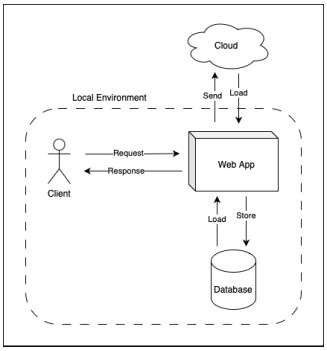

Figure 2: Local First schema

As depicted in [Figure 2], the client web application will have three essential tasks: client request management, fault tolerance, and cloud connection. For improved management and isolation of each action, *Worker Threads* with *Mutex* will be utilized. Since they will be manipulating the same data structure (a CRDT [2], to be further explored), it is necessary to ensure concurrency control and inhibit potential errors and inconsistency.

2.1 Client Request Management

This includes handling user inputs, processing requests promptly, and ensuring a responsive interaction with the application.

2.2 Fault tolerance

The web application will periodically store the volatile manipulated information in the local database file. This way, even if there is an error in the application or connectivity issues, most, if not all, of the user's changes will be saved, including those that have not yet been backed up to the cloud.

2.3 Cloud connection

In general, the previously described approach will ensure the proper functioning of the web application even without a connection to the cloud, thus limiting the ability to back up or transfer information. To address this, the application will periodically attempt to establish a connection with the cloud for the following purposes:

- Updating local information based on what is received from the cloud;
- Sending modified local information to the cloud to propagate it throughout the system.

3 CLOUD

In this cloud-based system, clients connect to a central proxy server. This architecture offers several key advantages:

- An end-to-end system without the user being aware of the cloud implementation, including details such as the number of available servers or their corresponding addresses;
- Elimination of the need for a fixed connection between the client and server or a fixed number of servers always available;

The implemented proxy server serves a critical additional function: load balancing [4]. Load balancing is essential to prevent performance degradation or bottlenecks when handling extensive requests on a single server, ultimately enhancing the efficiency of the entire system. To achieve load balancing, we employ the ZeroMQ library, utilizing ROUTER-REQ connections in both the frontend (client-proxy connection) and the backend (proxy-server connection).

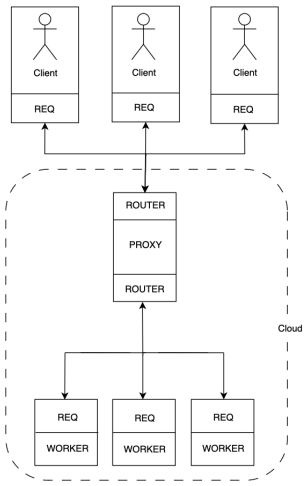

Figure 3: Proxy Schema

The core of the solution lies in the strategic management and distribution of data. The proposals for replication and sharding are directly informed by the architecture of Amazon Dynamo [1], providing a concrete and proven strategy for achieving scalability and resilience.

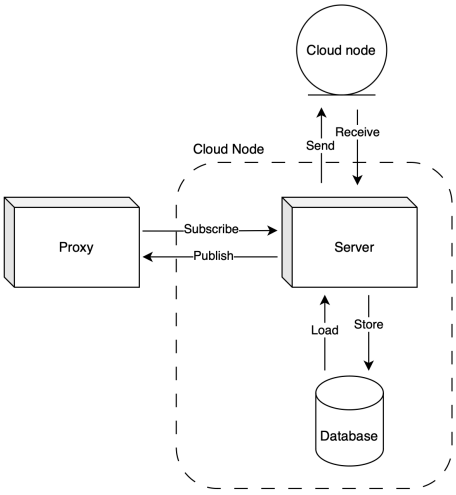

Figure 4: Cloud Node

As stated in [Figure 4], the server-side application will also have three threads, with proper concurrency control, to perform essential tasks: client request management, fault tolerance, and replication.

3.1 Client Request Management

When the proxy redirects requests to the server, the server is responsible for adjusting its internal Conflict-free Replicated Data Type (CRDT) [2] based on client updates. The response to the request will be another CRDT whose content reflects the current state of the system for the lists known to the client.

3.2 Fault Tolerance

Just like on the client side, there is a need for each node/server to have its own database. Therefore, the server side web application will periodically store the volatile manipulated information in the local database file.

3.3 Replication between nodes

To ensure eventual consistency across the entire system, the replication of modified data between servers is crucial. Upon instantiation, servers gain access to a list of neighboring servers. The order of each server's list enables the construction of a dependency network in the form of a ring, as illustrated in [Figure 5].

When a node detects a modification in its internal CRDT, it proceeds to communicate and propagate this alteration to the N neighboring nodes, following the specified order. The propagation involves a merge between the CRDTs of the two parties. In terms of connection fault tolerance, if a server among the chosen N servers does not respond, it is skipped, and communication is redirected to another remaining server.

An interruption or failure of a node does not signify a permanent exit from the ring; therefore, it should not result in the rebalancing of the assignment of these partitions.

4 CRDT

CRDTs (*Conflict-free Replicated Data Types*), will be utilized for message exchange both between client-server and among server nodes. These data types offer a distinctive approach to addressing consistency in distributed systems, enabling automatic convergence of replicated data, even in the presence of concurrent operations and asynchronous communication between nodes.

Given that each user should be able to instantiate and delete lists, as well as instantiate and delete items within each list, implementing CRDT for these data structures based on *ORMap* (*Observed Remove Map*) and *Enable Wins* is a prudent choice. Indeed, considering the project's context, concurrent deletions and updates of a list or item should promote the persistence of that structure in the system, thereby avoiding information loss.

To increment the quantity purchased for each item, the *GCounter CRDT* is the suitable choice as it efficiently handles concurrent increments.

All operations performed on these Delta-enabled CRDTs [3] are idempotent, ensuring the convergence of the current system state and eventual consistency.

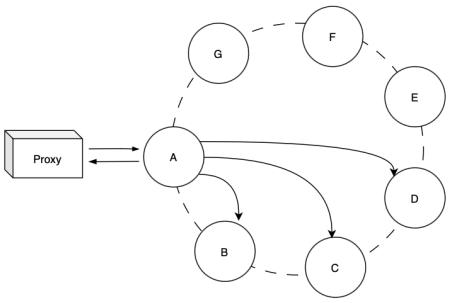

Figure 5: Replication Ring

REFERENCES

[1] *Amazon Dynamo*. https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.
[2] *CRDT*. https://crdt.tech/papers.html.
[3] *Delta Enabled CRDT*. https://github.com/CBaquero/delta-enabled-crdts.
[4] *Load Balancer*. https://zguide.zeromq.org/docs/chapter3/#The-Load-Balancing-Pattern.
[5] *Local First*. https://www.inkandswitch.com/local-first/.
[6] *UUID*. https://www.npmjs.com/package/uuid.
[7] *Worker Threads*. https://nodejs.org/api/worker_threads.html.
[8] *ZeroMQ*. https://github.com/zeromq/zeromq.js#examples.