Shopping List Application

Large Scale Distributed Systems - T01LIDL

Bárbara Rodrigues
up202007163

Nuno Silva
up202005501

Sofia Costa
up202300565

Tiago Ribeiro
up202007589

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Problem Description

Create Shopping List Application

Runs in the user device, can persist data **locally**, and also has a **cloud** component to share data among users and provide backup storage.

Users can create a new shopping list via the user interface.

Users who know the shopping list ID are always be allowed to add and delete items to the list.

Each item is associated with a **flag**, checked after the item is acquired.

1. **Use Last-Writer-Wins with local clocks.**

OR

2. **Use Conflict-free Replicated Data Types (CRDTs)**

Users can concurrently change the list.

Aiming for high availability.

Explicitly handle the replication of the shopping lists, including the consistency of the replicas.

Carefully design cloud-side architecture.

Aiming for millions of users.

Avoid data access bottlenecks.

Each list is independent and data can be sharded.

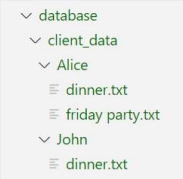
System Architecture

- **Client:**
  - Contains the logic for user interactions, managing local copies of data and invoking CRDT functions.
- **Server/"Cloud":**
  - Handles incoming requests from clients.
  - Executes the merging logic for resolving conflicts and synchronizing data between different client replicas.
- **Local Database:**
  - Ensures clients shared lists are not lost when the server goes down.
- **Load Balancer:**
  - Efficiently distribute incoming network traffic across multiple servers.

Client vs. Server

Client:

- Client initializes application.
- Identifies himself by provide username.
- Enters the list ID they wants to manage.



  - If it's a new list, it stores it in their personal computer, simulated as a folder with the client name in the database.
  - If the list was previous defined, it starts by getting the list's existing content, if any.
- Client will automatically try to synchronize with the Server after any list modification, but it can also choose to try to connect before making any list modification.
- Client only communicates with Server to send and receive list items, after this it closes the connection.

Server:

- A load balancer starts a group of servers and works to direct the clients to those.
- After Server is running it first checks if there are any stored lists, and loads its content if exists.
- After a Client communication, it merges the items received with the items that are stored in the respective list, if any
- When all the changes are finished it sends the updated list back to the Client and stores it in the database for future recovery if the Server goes down.

Load Balancer

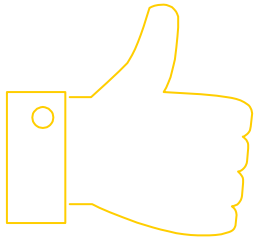
The **load balancer** is a critical component in distributed systems to efficiently distribute incoming network traffic across **multiple servers**.

When the Client sends their message, the identifier of the list they want to access is used to determine the Server the user will connect to. This way the load of requests is divided throughout these.

The Server each list belongs to is calculated by hashing the id with the **SHA-256 hashing algorithm**, then this hexadecimal string is converted into an “int”. Every Server is responsible for an equal interval of these “int” codes.

With the Server established, the load balancer forwards the message from the Client to the Server and the response from the Server back to the user, closing the connection after.

In case of a **high CPU usage**, the load balancer will create more servers, relieving the pressure on the other ones by thinning the interval of id codes they receive.



Conflict-Free Replicated Data Types

CRDT (1/2)

AWOR Map

Add Wins Observed Remove Map

- Used to facilitate the **addition** and **removal** of shopping list **items**.
- Enables users to add items simultaneously without conflicts and allow multiple users to contribute to a shopping list concurrently.
- Resolves conflicts when two or more replicas concurrently modify the same item (changing quantity or update acquired status).
- Uses **dots** to determine the most recent update or selecting one update randomly when the dot's timestamps are equal.

CRDT (2/2)

PNCounter & GCounter

Positive-Negative Counters & Grow-Only Counter

- Manage the **quantity** operations of shopping list items:
  - Handle simultaneous incrementing or decrementing operations, ensuring accurate quantity tracking despite concurrent modifications by multiple users.
- Manage **acquired status** operations of shopping list items.
  - Handle concurrent updates to the acquired status by multiple users, ensuring consistency in tracking whether an item has been acquired or not.

ShoppingList class

Functions:

- **init()**
  - Initializes the shopping list and its contents:
    - **self.id** (List identificator chosen by client)
    - **self.replica_id** (Client identificator)
    - **self.v** (Dotted Version Vector for operation timestamps, initialized at 1)
    - **self.shopping_map** (key = item id, value = item dictionary {name, quantity, acquired, timestamp}
    - **self.quantity_counters** (Stores the changes done to an item's quantity)
    - **self.acquired_counters** (Stores the changes done to an item's acquired status)
- **delete_list** (Deletes list and everything associated with it)
- **add_item** (Adds item to shopping map. An item is defined by a generated ID, name, quantity, status, timestamp)
- **increment_counter** (Increments the timestamp of a dot in the version vector)
- **remove_item** (Deletes item from list)
- **increment_quantity** (Increments selected item's quantity by 1 and updates the quantity_counters)
- **decrement_quantity** (Decrements selected item's quantity by 1 and updates the quantity_counters)
- **update_acquired_status** (Sets acquired status flag as True or False and updates the acquired_counters accordingly)
- **merge** (Deals with the possible add/remove item as well as quantity and acquired counters and merges the changes based on most recent update and final counter status)

PNCounter & GCounter

External classes applied to our application.

PNCounter used functions:

- **init(self, item_id, replica_id)**
  - self.P = GCounter(item_id, replica_id)
  - self.N = GCounter(item_id, replica_id)
  - self.item_id = item_id
  - self.replica_id = replica_id
- **inc(self, item_id, quantity=1)**
  - The function to increment the key's value in payload.
- **dec(self, item_id)**
  - The function to decrement the key's value in payload.

GCounter function used in the inc() and dec() functions of PNCounter:

```
def __init__(self, item_id, replica_id):
        self.payload = {}
        self.item_id = item_ id
        self.replica_id = replica_id


def inc(self, item_id):
    try:
        self.payload[item_id] += 1
    except Exception as e:
        print("{}".format(e))
```

Conclusions & Limitations

Conclusions

- The solution has met the project's goals in terms of creating a shopping list application and allowing multiple users to concurrently change the same list if they know the list ID.
- The concurrent changes that may result in merge conflicts are dealt with successfully, providing a conflict free application.
- The implementation of a load balancer was successful, allowing the efficient distribution of incoming network traffic across multiple servers.
- The project provided insights into CRDT methodologies, enhancing our understanding of distributed systems and conflict resolution strategies.

Limitations

- The acquired status doesn't work has expected, because for some reason when an item is acquired the PNCounter associated with it increments to 2 instead of 1 and -1 instead of 0.

Future Improvements

- Optimization of conflict resolution strategies
- Additional functionalities such as user authentication
- Improve error handling

DEMO

CLI

Editing Items

Thank you!

Any questions?