SHOPPING LISTS ON THE CLOUD

Fábio Morais – up202008052

Francisco Prada – up202004646

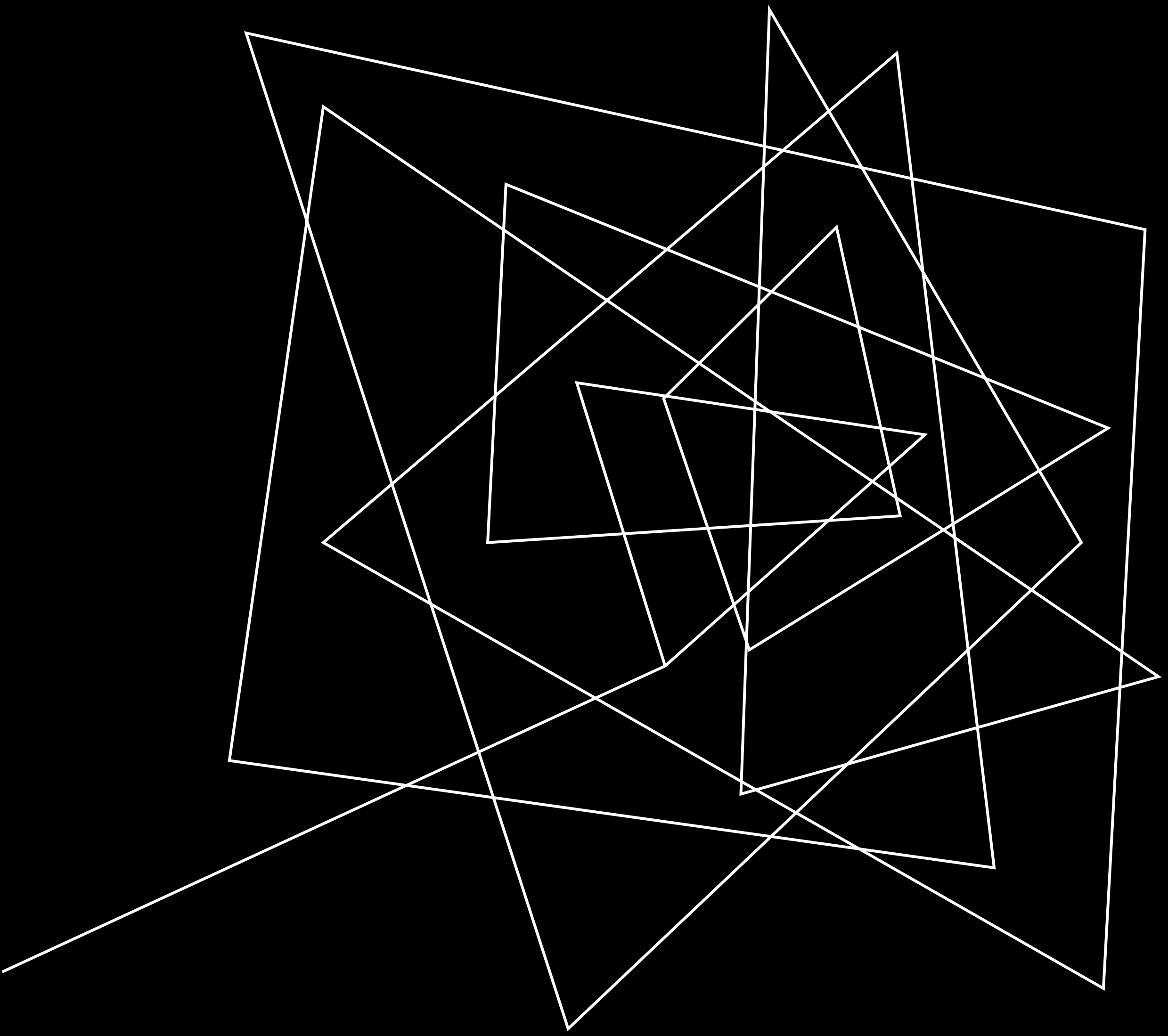
Guilherme Sequeira – up202004648

Pedro Ramalho – up202004715

INDEX

REQUISITES

- Local-first Shopping List Application;
- Shopping List Creation and Sharing;
- Local Persistence and Cloud-based component;
- Item management and quantity;
- Synchronization with CRDT’s;



TECHNICAL SOLUTION

CLIENT

By launching the application, the client will be able to:

- Insert their username.
- This procedure allows us to simulate clients in separated computers.

Then:

- **Open** a local list.
- Download a list from the cloud (**pull**).
- **Create** a new shopping list.

Once inside the 'Edit Shopping List' window

- The user can **add** new items.
- **Increment/decrement** their quantity.
- Synchronize the local shopping list by **pulling** the cloud version and joining them.
- Upload their shopping list to the cloud (**push**).
- **Save** their shopping list to their local storage.

CRDT'S - PRIORITIES

On this implementation of the CRDT's, there are certain aspects that should be considered when analyzing their operations:

- **Add-Wins and Remove Operations:** In scenarios where an object is deleted in one replica and subsequently re-added in another, the added object takes precedence over the deletion. However, it's important to note that during this process, any attributes or metadata associated with the object get completely reset and lost as the deletion interprets it as a different object (deletion wins over operations).
- **History of Values:** When examining the changes made to an object across different replicas (e.g., replicas A and B), the CRDT maintains a record of alterations made to the value of the object. This record is quite important as, when we merge both with each other, we only get the operations that we still didn't have access (if we have already merged with a previous version of that CRDT, we will only get the new operations).

KEY-VALUE STORE

A Key-Value store is preferred over a standard relational database:

- A single entity in our system: the shopping lists.
- Intended to store relatively small objects.
- Large number of *READ/WRITE* operations.

Supports two main operations: *get()* and *put()*

- *get()* returns the list with the given URL.
- *put()* updates the contents of the shopping list with the given URL or creates it if it doesn't exist.

CRDT'S

ORMap

This is the map CRDT that will store all the necessary data to be sent to the cloud as a JSON object.

This CRDT is composed of:

- A **map** whose entries will have as a key the name of the object (ex: banana) and the corresponding **CCounter**.
- A custom **ORMapHelper** responsible for keeping track of the object creation via **dot**.
- An **ID**, which is a string that identifies the replica.

CCounter

This is a causal counter CRDT, which allows to keep causality between its different values and has two distinct attributes :

- A **DotKernel** CRDT, which serves as a basis for the whole **CCounter** implementation.
- The **ID** of the replica.

DotKernel

The DotKernel is used as an auxiliar for the CCounter, using dot-based causality tracking.

It has two main components:

- A **DotContext** CRDT, which is crucial for the selection of dots that need to be processed.
- A **DotMap**, responsible for mapping the **dots** of the object and its value. These dots are a pair of the ID of their replica and a sequential **sequence number**.

DotContext

This context CRDT is indispensable when it comes to keep up with all the operations done to the object. It is divided into two main components:

- A **CausalContext** map, which keeps track of highest sequence number for the replicas that were already seen.
- A **DotCloud** set, used to keep track of the dots which **can't** yet be processed since there are still **dots missing** before them.

LOAD BALANCER

The Load Balancer acts as an intermediary between the Client and the Nodes.
The other parts of the application interact with the Load Balancer like so:

Client

- Requests *READ* and *WRITE* operations
- Polls the status of said operations, which can be:
  1. PROCESSING
  2. ERROR
  3. DONE
- Fetches the result of *READ* operations, after making sure the request is DONE.

Nodes

- Receives *READ* and *WRITE* requests.
- Updates the status of received requests.

LOAD BALANCER

Client sends a *READ* or *WRITE* request → Unique request ID is generated and returned to the Client → Load Balancer sends a *READ* or *WRITE* request to a random node including the request ID

(Asynchronously)

Nodes

Node receives request

↓

(Internal logic...)

↓

Node sends an update to Load Balancer

Client

Client receives request id

↓

Polling until DONE, ERROR or *Timeout*

↓

Client updates UI if needed

LOAD BALANCER

To support the specified architecture, an HTTP server is launched with the following endpoints:

- ***GET* /read/{ID}** – Receives *READ* requests of the Shopping List whose identifier is **ID**.
- ***PUT* /write/{ID}** – Receives *WRITE* requests of the Shopping List whose identifier is **ID**.
- ***PUT* /nodes/read/{FORID}** – Receives a status update for a *READ* operation with request id **FORID**.
- ***PUT* /nodes/write/{FORID} –** Receives a status update for a *WRITE* operation with request id **FORID**.
- ***GET* /client/poll/{FORID} –** Receives a poll request to query the state of *READ* or *WRITE* request with request id **FORID.**
- ***GET* /client/read/{FORID} –** Receives a fetch request for the result of a *READ* operation with request id **FORID**.

NODE SERVER

SERVER DISCOVERY

- There is a hardcoded number of **Seed** servers which all Node Servers know *a priori* and are assumed to be running.
- The **Seed** servers are responsible for always knowing what servers have been added and/or removed and are updated manually every single time a server is added/removed.
- When a node is started, it starts by querying the available **Seed** servers, thereby gaining knowledge of available servers.
- Whenever an internal request between nodes is done, their view of available servers is also exchanged, and the most recent one is kept.

NODE SERVER

CONSISTENT HASHING

- Node Servers are organized in a **ring**.
- A single Node Server contains multiple **Virtual Nodes** in the ring.
- The **number** of Virtual Nodes can be customized per Server.
- The placement of the Virtual Nodes in the ring (**token**) of a given Server is *random*, but **deterministic**
  - Each position of a Virtual Node is obtained by hashing the identifier and port of the server suffixed by the index of the virtual node (0, 1, 2...).
- For each list, its token is calculated using its identifier (URL).
- The token is then used to find a given number of **unique** servers in which the list **can** be replicated.
  - This is called the list's **priority list**. Its length is a system-wide configuration, but it should be bigger than the also system-wide Replication parameter, **N**.

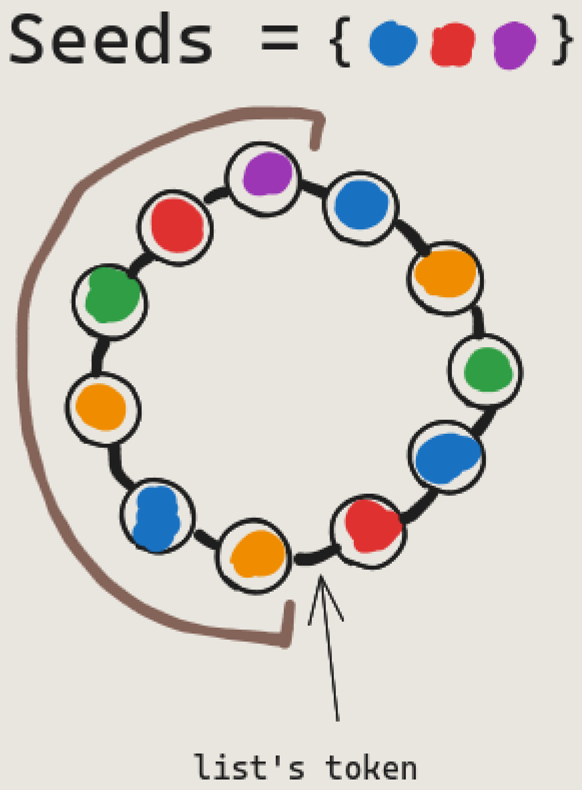
NODE SERVER

INTERNAL REQUEST LOGIC

- When a *READ* or *WRITE* request for a list is issued, a random node is tasked with the operation, becoming its **Coordinator**, although a READ request is forwarded to a node in the priority list in case the original **Coordinator** is not a part of it.
- The first N available nodes in the list's priority list are issued an internal request.
- The **Coordinator** then counts the number of successful internal requests and approves or fails the operation based on system-wide parameters, **W** for writes and **R** for reads.
- The **Coordinator** then sends an update to the load balancer indicating the result of the operation (success or fail) and, in case of a READ operation, the resulting list, which is then also saved to the local storage of the **Coordinator**.

NODE SERVER

EXAMPLE REQUEST



System Parameters

- Priority List Length: 5
- N: 3
- R: 2
- W: 1

SOLUTION EVALUATION

MAIN POINTS AND LIMITATIONS

- The application supports the expected operations of a shopping list platform with a cloud component
  - Addition/removal of items.
  - Increment/decrement of their quantities.
  - Local storage with persistent storage in the cloud, simulating different computers for each user and server.
  - Synchronization of lists between users.
  - Consistent Hashing.
  - CRDT Shopping List implementation.
  - Data replication and multiple failure tolerance mechanisms, which prevent a single node failure from severely affection system operation and durability.

- Although, there are limitations
  - The logic for addition/removal of a node from the ring has been implemented, though it needs to be done manually by an external tool that has not been implemented.
  - Messages that update the user about the failure / success of read/write requests hasn't been implemented in the UI.
  - Hinted handoff has not been implemented.
  - The CRDT's implementation maintains a record of alterations for each replica it encounters, potentially resulting in significant space consumption.
  - Code documentation is limited.