BuyBuddy

SDLE Project Presentation

SDLE 23/24

- Carlos Baquero-Moreno
- Pedro Ferreira Souto

- André Costa, up201905916@up.pt
- Bárbara Carvalho, up202004695@up.pt
- Fábio Sá, up202007658@up.pt
- Luís Cabral, up202006464@up.pt

Index

Context and Overview

- **Local-First Design**: prioritizes a local-first approach, running code on user devices for data persistence. This ensures offline functionality, enhancing user experience by allowing them to access and modify their shopping lists without an internet connection.

- **Collaborative Lists**: each shopping list has a unique ID, facilitating collaboration. Users with the list ID can seamlessly collaborate, enabling them to collectively manage and update shopping lists.

Technology

We opted for user-friendly web application, prioritizing simplicity in technology choices for easy installation and use.

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

Additionally, for distributed system connections, cloud management, and maintaining integrity and consistency:

- `ZeroMQ`, for high-performance asynchronous messaging;
- `UUID`, for the generation of unique identifiers across the entire system;

Cart API

Each node in the distributed system instantiates and manipulates a Cart object that gathers CRUD (Create, Read, Update, Delete) functions and data consistency features.

Advantages:

- Functional for **any node** in the system (client or server);
- **Encapsulates** complex operations and algorithms for data consistency (**CRDTs**);
- It is the only variable that nodes have to manipulate, **simplifying concurrency control**;

```
// initialization
const cart = new Cart(nodeID);
cart.load(db);

// CRUD
const url = cart.createList('List A');
cart.deleteItem(url, 'Peras');
cart.updateItem(url, 'Lima', 19, 20);

// get information for frontend
cart.info();

// serialize object for messaging and
merge
cart.toString();

// synchronization
cart1.merge(cart2.toString());
cart2.merge(cart1.toString());
```

Cart API

- Cart contains a **Map** between the URL of each list and an AWORSet, enabling **constant-time data retrieval** for a list;
- **AWORSet** (*Add Wins Observed Remove Set*), a State-Based CRDT, **keeps track of all causal contexts** of the list but only retains in memory the **items that have not been removed**, improving **spatial performance**;
- **GCounter** (*Grow-only Counter*) manipulates the **total and partial quantities** of each item;

```
class Cart {
  // Map<url, AWORSet>
  this.lists = new Map();
}

class AWORSet {
  // [(itemName, GCounter, (nodeID, version))]
  this.set = [];

  // [(nodeID, version)]
  this.causalContexts = [];
}

class GCounter {
  this.currentValue = 0;
  this.totalValue = 0;
}
```

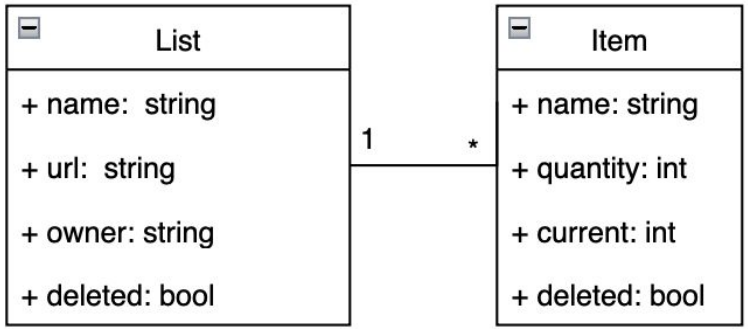
Local First

In the initial phase, the client app checks for the presence of a **local database** following the predefined schema;

The URL is instantiated locally using **UUID v4**:

- global uniqueness
- it does not rely on any user input, which may not be unique throughout the system
- is independent of timestamps, which can be desynchronized between nodes

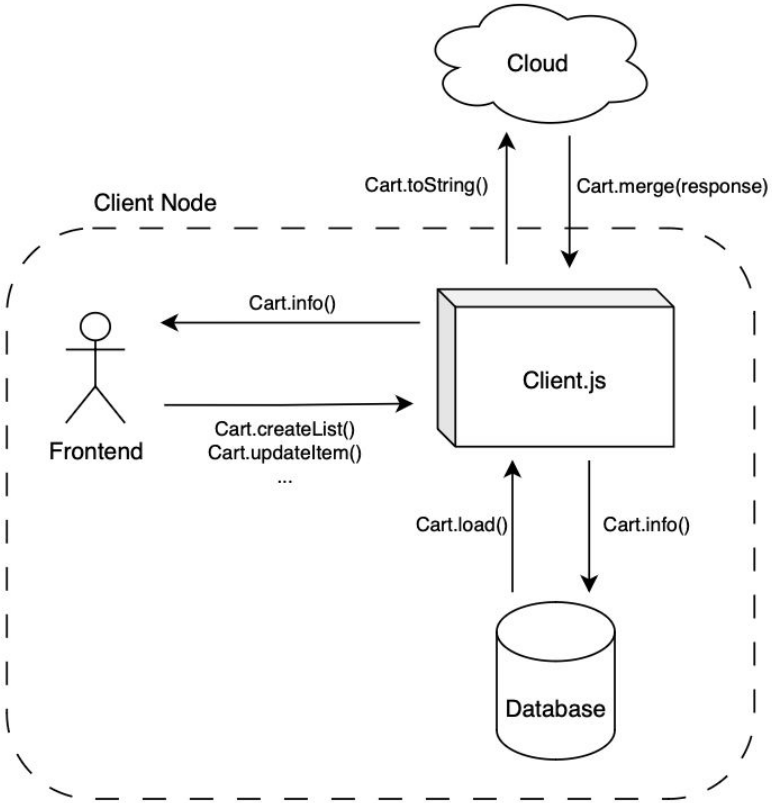
Only the owner of the list (the node that instantiated it) can delete it, and this operation is irreversible, propagating throughout the entire system.



Local First

- **Frontend request management**;
- **Fault tolerance**: periodically, the current state of the Cart is stored in the local database, allowing data recovery from the node in case of failure;
- **Cloud connection**: periodically, it tries to send the current state of the cart to the cloud, whether there is someone reading or not. If there is, the response, which is a sub-state of the server that the client should be aware of for the update, can be merged immediately.

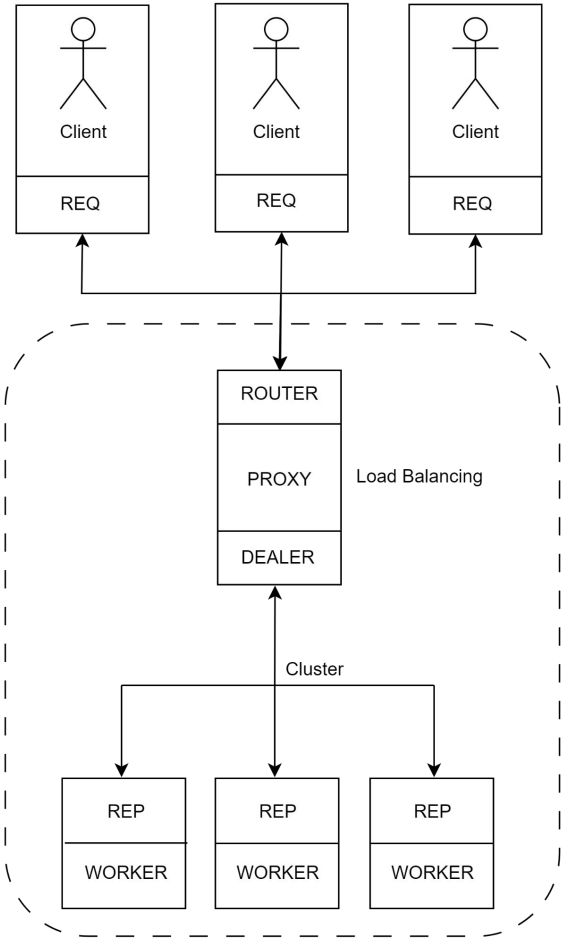
Improved management and isolation of each action by using **Worker Threads**, with **Locks** for Cart concurrency control.



Cloud

The cloud infrastructure is implemented using the ZeroMQ library in Node.js, facilitating communication among clients and servers. The system follows a **Request-Router-Proxy-Dealer-Reply** architecture.

- **Clients**: Utilize a Request socket to connect to a Router socket.
- **Router**: Acts as an entry point for clients and connects to a Proxy socket.
- **Proxy**: Interconnects the Router socket with a Dealer socket, enabling distributed work allocation.
- **Dealer**: Distributes work among multiple server instances using the cluster library, ensuring **Load Balancing**.
- **Servers**: Employ a Reply socket to respond to the requests, completing the Req/Rep communication pattern.



Cloud

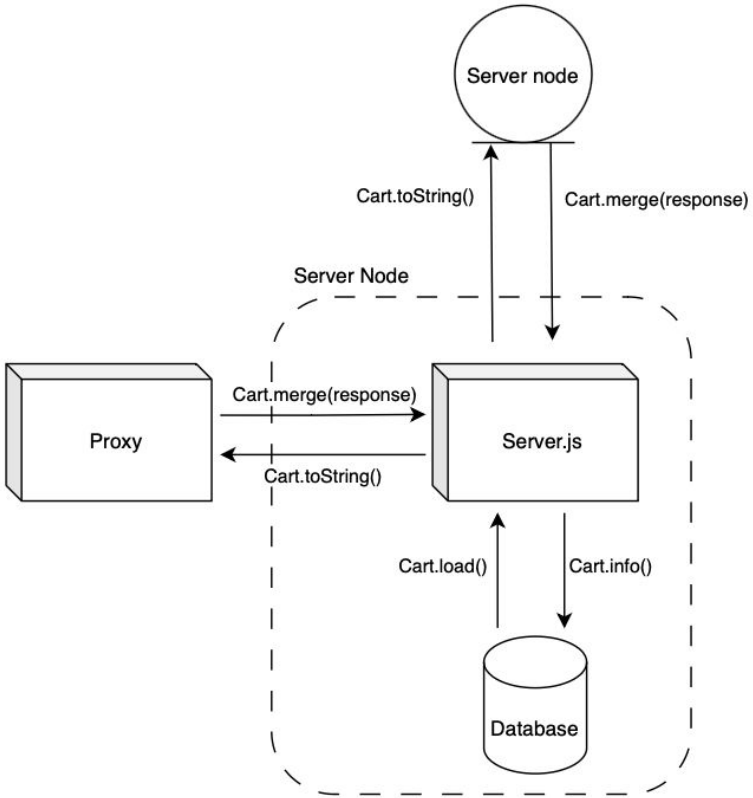
Key Features:

- **End-to-End System**: Clients interact with the system without being aware of the underlying cloud details, such as the number or addresses of servers.
- **Dynamic Connection Handling**: Eliminates the need for a fixed connection between clients and servers. The system adapts to varying server availability, ensuring flexibility.
- **Server Recuperation**: Supports automatic recovery of server nodes in case of failure, enhancing system robustness.
- While server recuperation is supported, the system requires a reboot if the Proxy fails.

Cloud

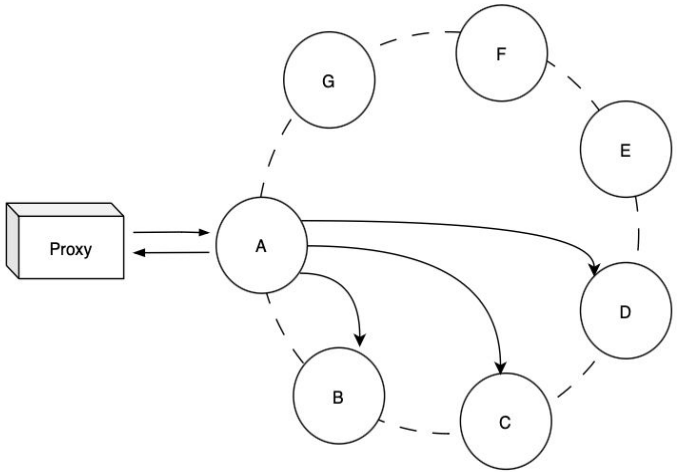
- **Client request management:** Receives the current state from the client, merges it, and returns the relevant sub-state to the client;
- **Fault tolerance**: periodically, the current state of the Cart is stored in the local database, enabling data recovery from the node in case of failure;
- **Ring replication**: each server can periodically send its state to another neighboring server or receive updates from its neighbors;

Improved management and isolation of each action by using **Worker Threads**, with **Locks** for Cart concurrency control.



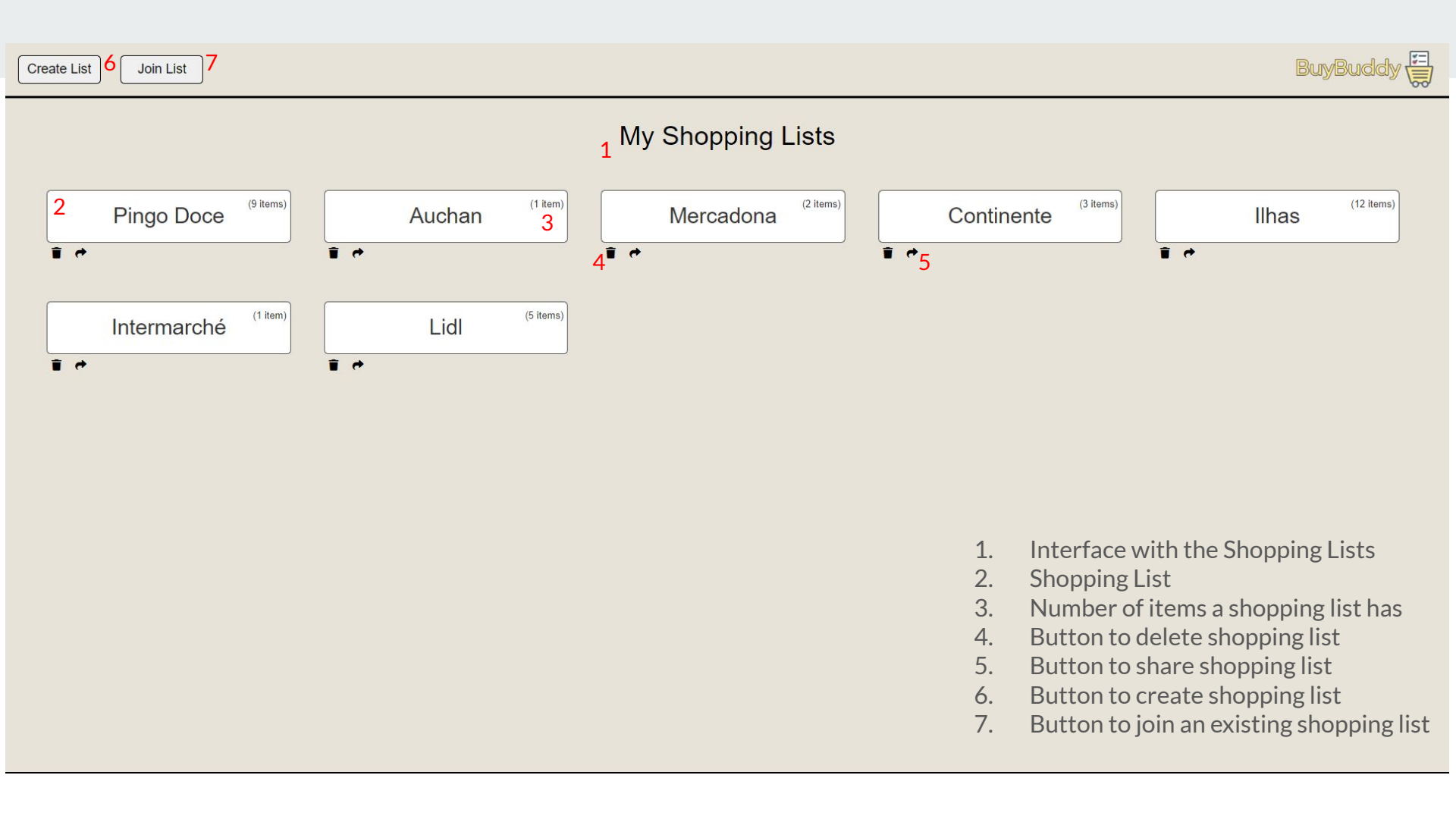
Cloud

- To ensure eventual consistency across the entire system (lists sharding)
- 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;
- Periodically, it proceeds to communicate and propagate this alteration to the N neighboring nodes, following the specified order.
- 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.

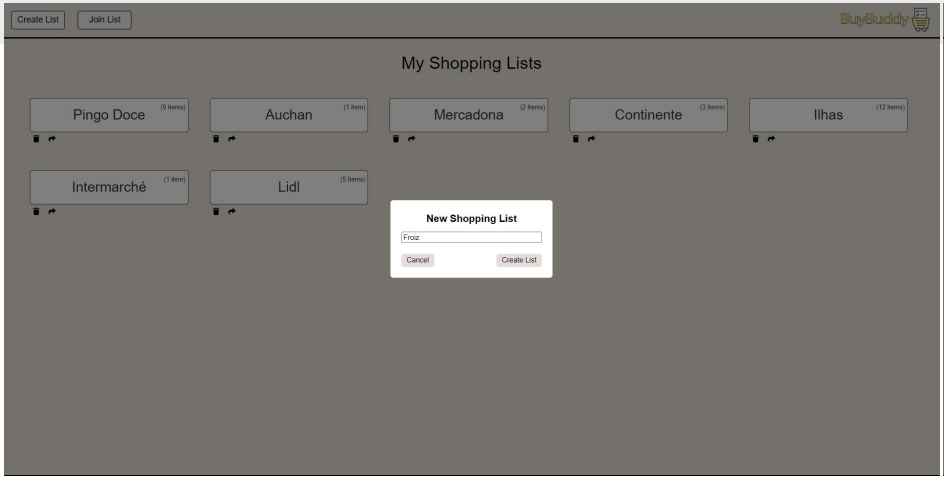


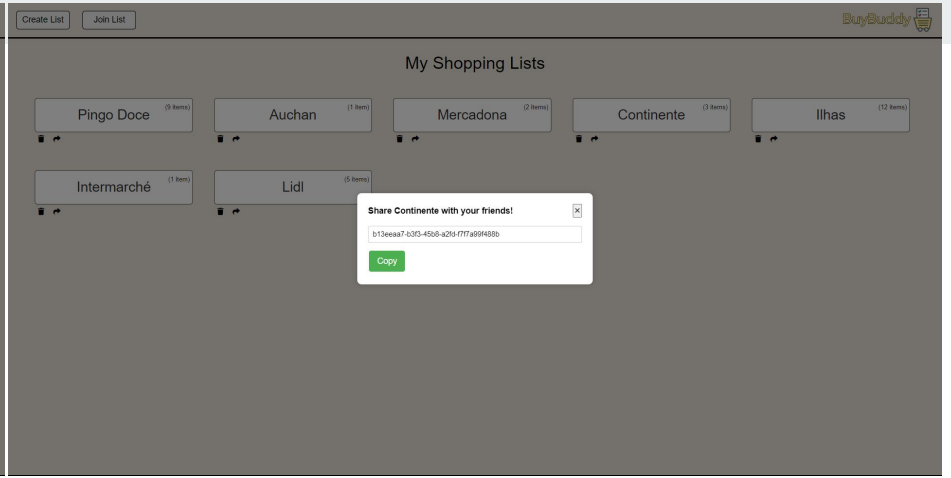
User Interface

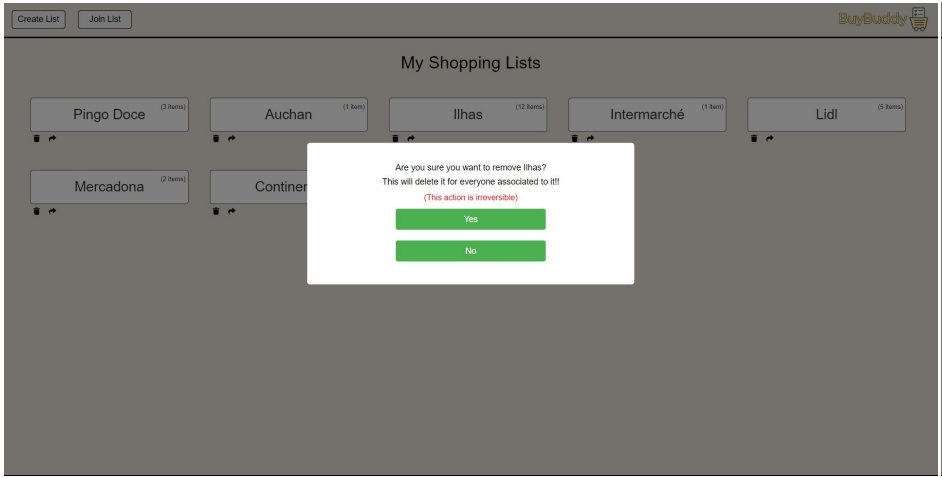
- We decided on a minimalist interface with buttons for the main features
- This helps the user feel comfortable with the experience, as well as allowing different kinds of users to fully enjoy the extent of BuyBuddy
- Each action can be performed by pressing a sequence of buttons on the screen, and occasionally type text into the respective fields.
- The user's shopping lists are shown on the screen under the form of buttons.

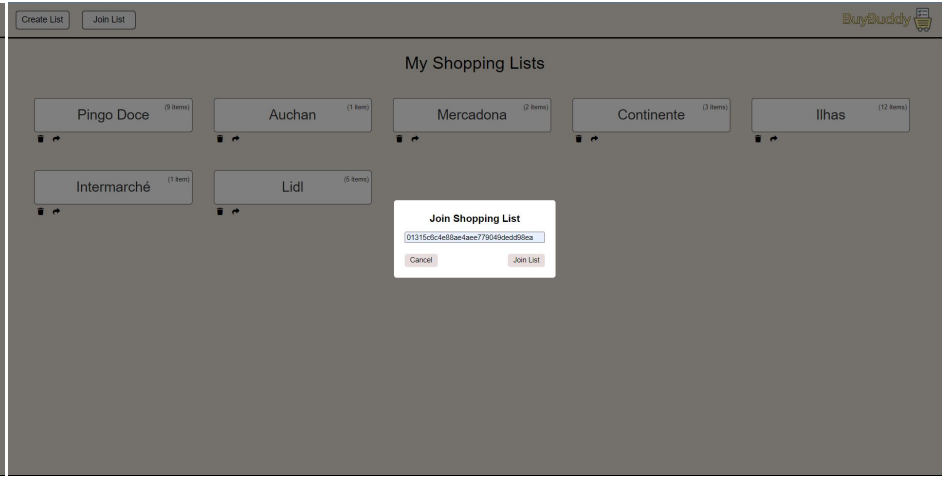


1. Interface with the Shopping Lists
2. Shopping List
3. Number of items a shopping list has
4. Button to delete shopping list
5. Button to share shopping list
6. Button to create shopping list
7. Button to join an existing shopping list


Create List
Join List
BuyBuddy
My Shopping Lists
Pingo Doce (9 items)
Auchan (1 item)
Mercadona (2 items)
Continente (3 items)
Ilhas (12 items)
Intermarché (1 item)
Lidl (5 items)
New Shopping List
Froiz
Cancel
Create List


Create List
Join List
BuyBuddy
My Shopping Lists
Pingo Doce (9 items)
Auchan (1 item)
Mercadona (2 items)
Continente (3 items)
Ilhas (12 items)
Intermarché (1 item)
Lidl (5 items)
Share Continente with your friends!
b13eeaa7-b3f3-45b8-a2fd-f7f7a99f488b
Copy

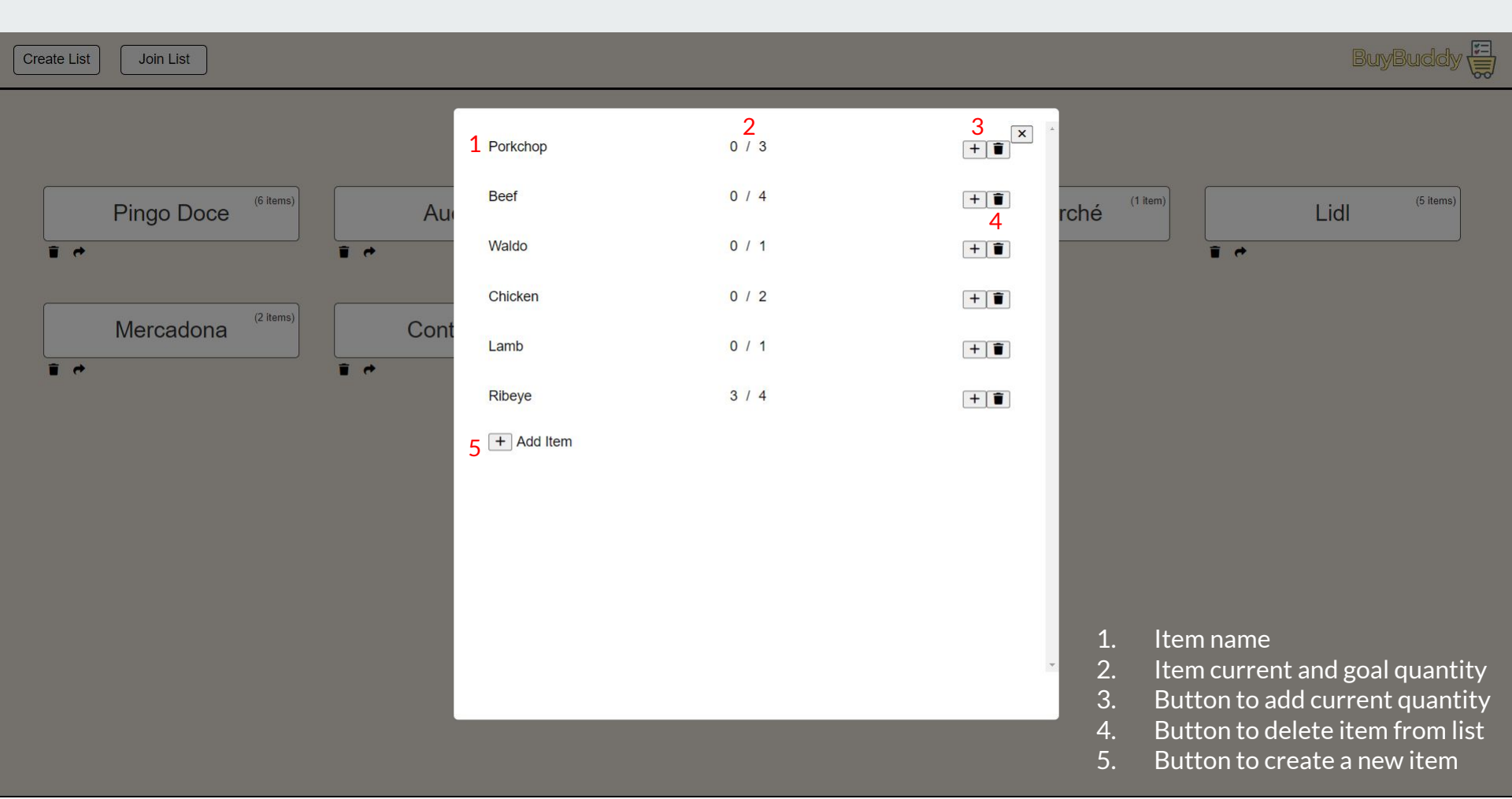

Create List
Join List
BuyBuddy
My Shopping Lists
Pingo Doce (3 items)
Auchan (1 item)
Ilhas (12 items)
Intermarché (1 item)
Lidl (5 items)
Mercadona (2 items)
Are you sure you want to remove Ilhas?
This will delete it for everyone associated to it!!
(This action is irreversible)
Yes
No


Create List
Join List
BuyBuddy
My Shopping Lists
Pingo Doce (9 items)
Auchan (1 item)
Mercadona (2 items)
Continente (3 items)
Ilhas (12 items)
Intermarché (1 item)
Lidl (5 items)
Join Shopping List
01315c6c4e88ae4aee779049dedd98ea
Cancel
Join List

User Interface

- We decided to let the User manipulate every change before actually submitting them, so GCounter is suitable for the needs of partial and total quantities updates.

- This strategy allows for a more User-Friendly experience, and prevents User errors to be inadvertently loaded into the System due to erroneous operations.

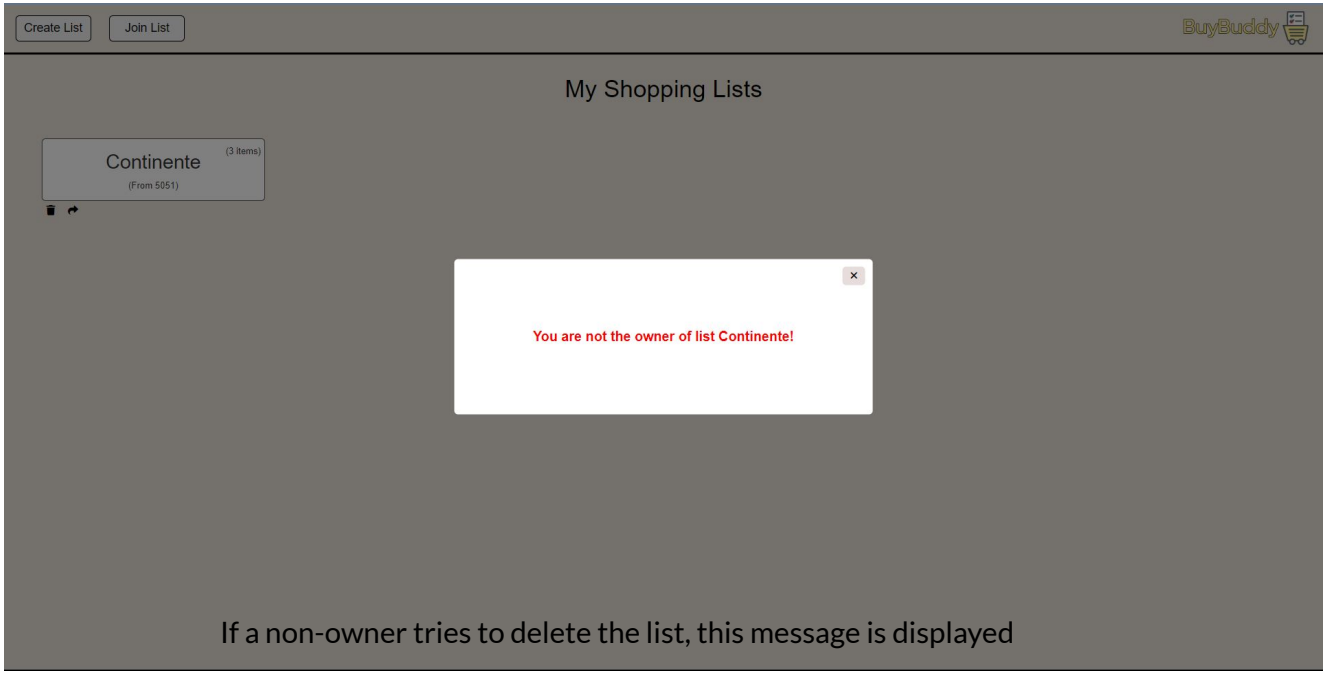


1. Item name
2. Item current and goal quantity
3. Button to add current quantity
4. Button to delete item from list
5. Button to create a new item


Create List
Join List
BuyBuddy
Pingo Doce
(6 items)
Mercadona
(2 items)
Lidl
(5 items)
Porkchop 0 / 3
Beef 2 / 4
Waldo 0 / 1
Chicken 0 / 2
Lamb 0 / 1
Ribeye 3 / 4
Venison 0 / 3
Add Item
1
2
3
4
1. Deleted item
2. Button to undo deletion
3. Button to reduce current item quantity
4. Button to save changes

User Interface

- **When a list is shared with another user, it's displayed in a different way in order to offer more information to the end-user.**



If a non-owner tries to delete the list, this message is displayed

In this case, the <port> is displayed

Improvements

- **Why isn't the causal context of a list also stored in the local database?**

  We don't need. Merging causal contexts is simply a union of two sets, so if the node loses the causal context, it can easily **recover it in the first interaction with the cloud**.

- **How is the import of a list from the cloud implemented?**

  The client simply needs to **instantiate a list on their side** with the imported URL. The merge between two lists is designed to not only **merge the items** in the list but also **update their attributes** (owner name, list name, deleted flag, loaded flag) that were previously unknown. In the first interaction with the cloud, all the content of the imported list becomes known.

Improvements

- **Is the output of a server's merge always its entire state?**

  No. For efficiency reasons, the output of the server after its internal merge is **never its entire content** and depends on the requesting node:

  - **Client-server interaction**: the client only receives the subset of the cart state that matters to it, i.e., the **latest version of the lists it knows**.
  - **Server-server interaction**: the other server only **receives an 'ACK' to confirm receipt**. At this stage, there is no need to exchange the total content of the two servers since, being arranged in a ring, they will soon receive the content from their neighbors.

- **In any node (client or server), is the storage to the Cart local database always performed?**

  No. For performance reasons, storing information in the local database is **only done when the Cart detects an internal change**.

References

- Amazon Dynamo - https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- CRDT - https://crdt.tech/papers.html
- Delta enabled CRDTs - https://github.com/CBaquero/delta-enabled-crdts
- Load Balancer - https://zguide.zeromq.org/docs/chapter3/#The-Load-Balancing-Pattern
- Local First - https://www.inkandswitch.com/local-first/
- UUID in JS - https://www.npmjs.com/package/uuid
- Worker Threads - https://nodejs.org/api/worker_threads.html
- ZeroMQ.js - https://github.com/zeromq/zeromq.js