

Listify

SDLE Project G07

Lucas Faria – up202207540
Alexandre Lopes - up202207015
Pedro Borges - up202207552

Index

- Project Context.
- Technologies.
- CRDT Implementation.
- Local First.
- Cloud.
- User Interface
- Main Challenges.

Project Context

- In applications serving millions of users, scalability and performance are essential to ensure a seamless experience with minimal bottlenecks. A local-first architecture allows users to continue interacting with the app even when disconnected from the cloud, preserving usability in offline scenarios.
- Yet, extending a local-first approach to collaborative features, such as shared shopping lists, introduces significant challenges, particularly in conflict resolution when merging data across multiple users.
- This project addresses these challenges by enabling the creation and sharing of shopping lists within an application designed to scale to millions of users. The system provides reliable data replication, effective conflict management, and strong safety guarantees, ensuring resilience, consistency, and a smooth collaborative experience.

Technologies

- The project relies primarily on web technologies to deliver a simple and efficient user experience:
 - **React.js** – for building the user interface.
 - **Node.js** – for handling application logic, including both client-side and server-side processes.
 - **SQLite3** – as the local database system for offline storage.
 - **Express.js** – to manage HTTP communication between the UI and the client worker.
 - **Node WebSockets** – for communication between the client and the cloud service.
 - **UUID v4** – to generate unique global identifiers for shopping lists

CRDT Implementation

- CRDTs (Conflict-free Replicated Data Types) provide a robust mechanism for merging distinct causal histories without conflicts. They are particularly well-suited for collaborative applications, as they enable sharing lists between users while ensuring eventual consistency across replicas.
- The main components of the implemented CRDT logic include:
 - **ShoppingList** – the high-level structure representing user lists.
 - **Aworset** – a set implementation that resolves concurrent add/remove operations deterministically.
 - **DotContext** – tracks causal relationships between operations.
 - **DotKernel** – manages the underlying state and ensures correct propagation of updates.
 - **PNCounter** – supports increment and decrement operations while maintaining consistency.
 - **GCounter** – a counter that only allows increments, ensuring monotonic growth across replicas.

CRDT Implementation – ShoppingList

- Enables the main operations on a list.
- Includes an **Aworset** to save the items.
- Also includes a map with **itemName -> PNCounters** where the quantities of each product are stored.
- The main operations are:
 - **addItem(name, quantity)**: adds an item to the Aworset and its quantity to the map.
 - **removeItem(name)**: removes an item from the Aworset and from the quantity map.
 - **markBought(name, quantity)**: marks a certain item as being both with a certain quantity,
 - **updateQuantity(name, diff)**: update the quantity of a certain item.
 - **merge(otherList)**: merges two lists Aworsets and quantity maps.
 - **getItems()** : gets all the items in the list.
 - **getItemsForDisplay()**: gets all the items in a format used by the UI.
 - **fromJson(json)**: creates a new list from Json.
 - **toJson()**: returns the list in json format.

CRDT Implementation – Aworset

- The Aworset structure implements the functionality of an Aworset CRDT, providing deterministic resolution of concurrent add/remove operations.
- It supports operations to add, remove, read items, and merge Aworsets across replicas.
- It integrates a DotKernel, which manages causal histories and ensures that updates are correctly tracked and propagated.

CRDT Implementation – DotKernel

- The DotKernel is a core component of the CRDT logic, responsible for managing items together with their causal history. It works alongside the DotContext to ensure that operations remain conflict-free and consistent across replicas.
 - **Dot Store:** Maintains a mapping between dots (unique identifiers combining replica ID and counter) and items.
 - **Add Operation:** Creates a new dot for each item added and records it in both the local store and causal context.
 - **Remove Operation:** Deletes items by marking their associated dots as removed, ensuring consistency across replicas.
 - **Merge (Join):** Integrates the state of another replica, reconciling additions and removals while preserving causal order.
 - **Remove All:** Clears all items by marking their dots as removed.
 - **Serialization:** Provides toJson and fromJson methods to export and reconstruct kernel state for communication or persistence.

CRDT Implementation – DotContext

- The DotContext is a fundamental component of the CRDT system, responsible for tracking and managing causal histories of operations across replicas. It ensures that updates are correctly ordered, compacted, and synchronized, forming the backbone of conflict-free replication.
 - **Compact Causal Context (cc):** A map that records, for each replica, the highest counter value observed. This provides a compressed representation of causal history.
 - **Dot Cloud (dc):** A set of individual dots (replicaId:counter) that represent operations not yet compacted into the causal context.
- Key Operations in the implementation:
 - **dotin(dot):** Checks whether a given dot is already known, either in the compact context or the dot cloud.
 - **makedot(replicaId):** Generates a new dot for a replica by incrementing its counter, ensuring unique causal identifiers.
 - **insertDot(dot):** Adds a dot to the dot cloud and optionally compacts it into the causal context.
 - **compact():** Merges contiguous dots into the compact context, pruning redundant entries and keeping the state efficient.
 - **join(otherContext):** Synchronizes two contexts by merging their causal histories and dot clouds, ensuring consistency across replicas.
 - **Serialization:** Provides toJson and fromJson methods to export and reconstruct context state for communication or persistence.

CRDT Implementation – PNCounter

- The PNCounter extends the functionality of a GCounter to support both increments and decrements while maintaining eventual consistency across replicas. It achieves this by combining two grow-only counters:
 - **Positive Counter (p)**: Tracks all increment operations.
 - **Negative Counter (n)**: Tracks all decrement operations.
- Key Operations in the implementation:
 - **inc(amount)**: Increases the counter by a specified amount, updating the positive counter.
 - **dec(amount)**: Decreases the counter by a specified amount, updating the negative counter.
 - **local()**: Returns the local value of the counter ($p.local() - n.local()$).
 - **read()**: Computes the globally consistent value across replicas ($p.read() - n.read()$).
 - **join(other)**: Merges the state of another replica by synchronizing both positive and negative counters.
 - **Serialization**: Provides toJson and fromJson methods to export and reconstruct the counter state for communication or persistence.

CRDT Implementation – GCounter

- The GCounter is designed to support increment-only operations while ensuring eventual consistency across distributed replicas. Each replica maintains its own counter, and the global value is computed by summing all replica contributions.
 - **Replica Counters:** Each replica is identified by an ID and maintains its own local counter.
 - **Increment (inc):** Increases the local counter by a specified amount. The operation is monotonic; values can only grow.
 - **Local Value (local):** Returns the current value of the counter for the local replica.
 - **Global Value (read):** Computes the total value by summing all replica counters.
 - **Merge (join):** Synchronizes with another replica by taking the maximum value observed for each replica ID, ensuring no increments are lost.
 - **Serialization:** Provides toJson and fromJson methods to export and reconstruct counter state for communication or persistence.

Local First

- Initially, all data is stored locally in a database (schema visible in the image), and operations on the shopping lists are handled through local CRDTs. This enables users to perform every action offline, including:
 - **Creating new lists**
 - **Adding products**
 - **Updating quantities of required items**
 - **Marking products as purchased**
- Because these operations are managed locally, the application remains fully functional even without an internet connection. Sharing lists and synchronizing with cloud storage become possible only when a connection to the cloud service is available, ensuring that collaboration features are seamlessly integrated once connectivity is restored.



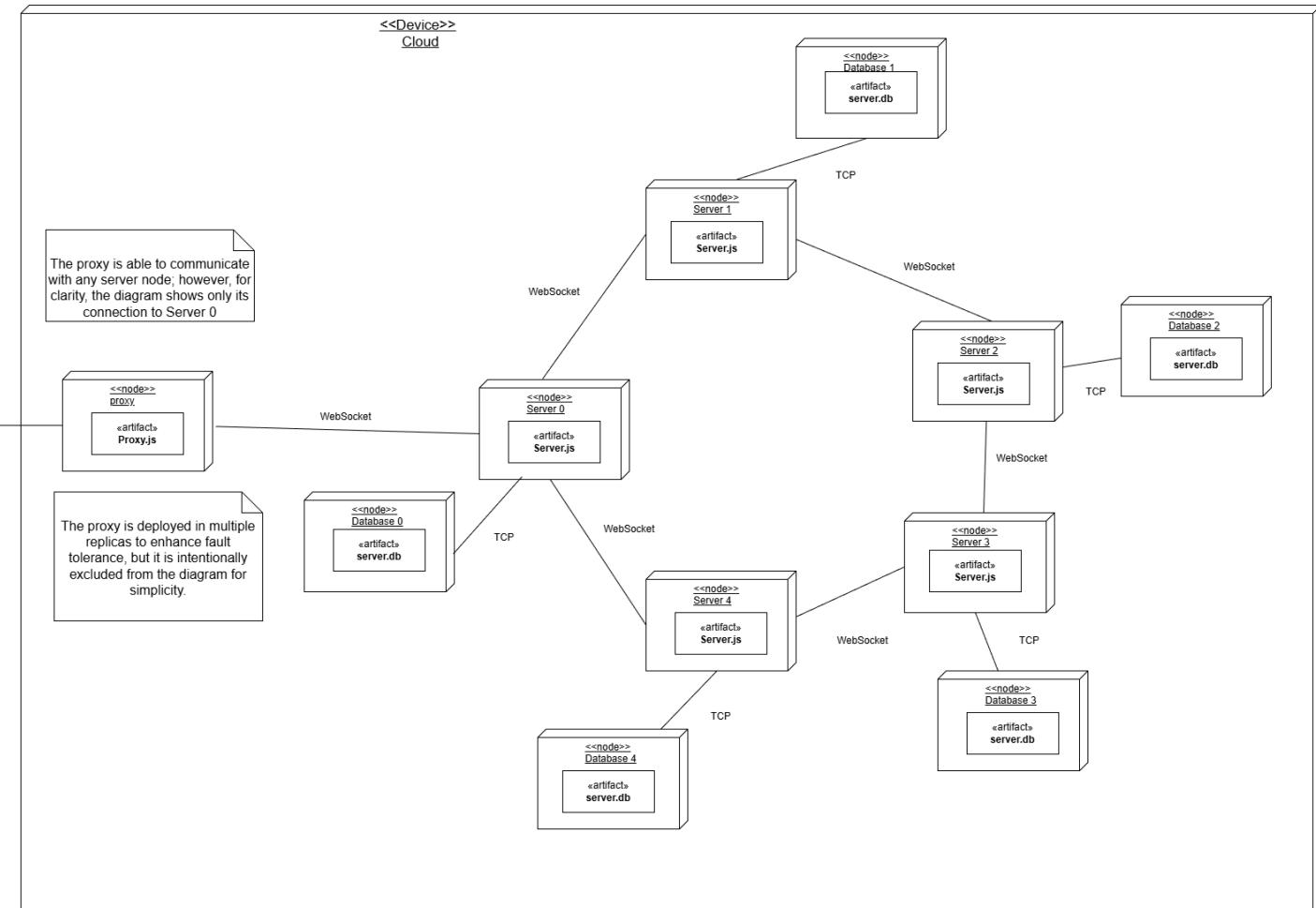
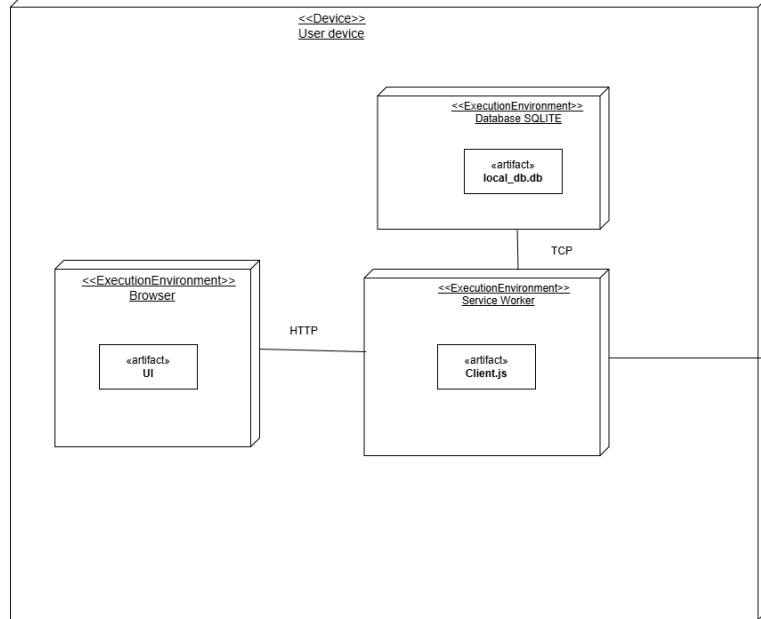
Cloud

- The cloud component of the project enables users to share shopping lists seamlessly across devices and with other users. The design follows an approach inspired by Amazon DynamoDB, leveraging distributed principles for scalability and fault tolerance.
 - **Ring Topology:** Cloud servers are organized in a ring structure, ensuring balanced distribution and resilience.
 - **Proxy Layer:** A proxy receives client requests and, using a consistent hashing algorithm (based on the DynamoDB paper), determines the appropriate server to handle each list.
 - **Global List ID:** The server assignment is calculated using the list's global identifier. If a list arrives without a global ID, the proxy generates a new one to guarantee uniqueness and consistency across the system.

Cloud - Replication

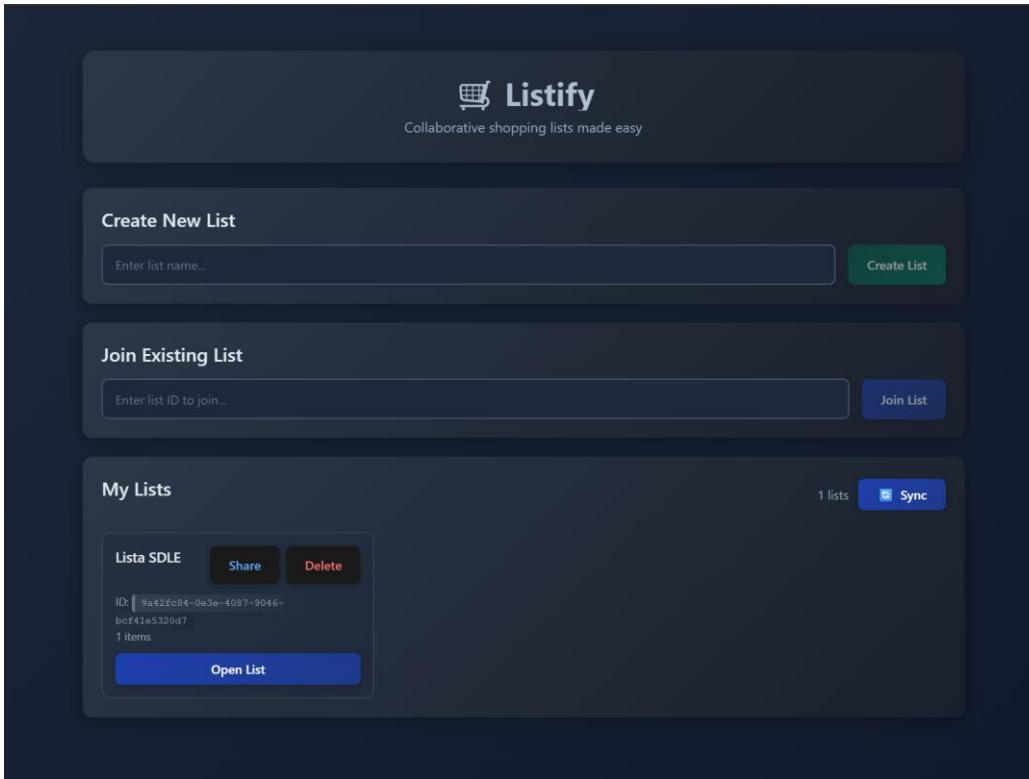
- Each server maintains N replicas (three in our implementation) to ensure durability and availability.
 - **Preference List:** The consistent hashing algorithm generates a preference list for each shopping list, ranking servers in order of responsibility.
 - The first server in the list acts as the primary, storing the authoritative copy of the data.
 - The remaining servers serve as backups, ready to take over if the primary becomes unavailable.
 - **Replication Process:** When a list is received, a worker on the primary server propagates the data to its replicas. This guarantees that the list is consistently stored across multiple nodes, reducing the risk of data loss and enabling fast recovery in case of failures.

UML Deployment Diagram of the System

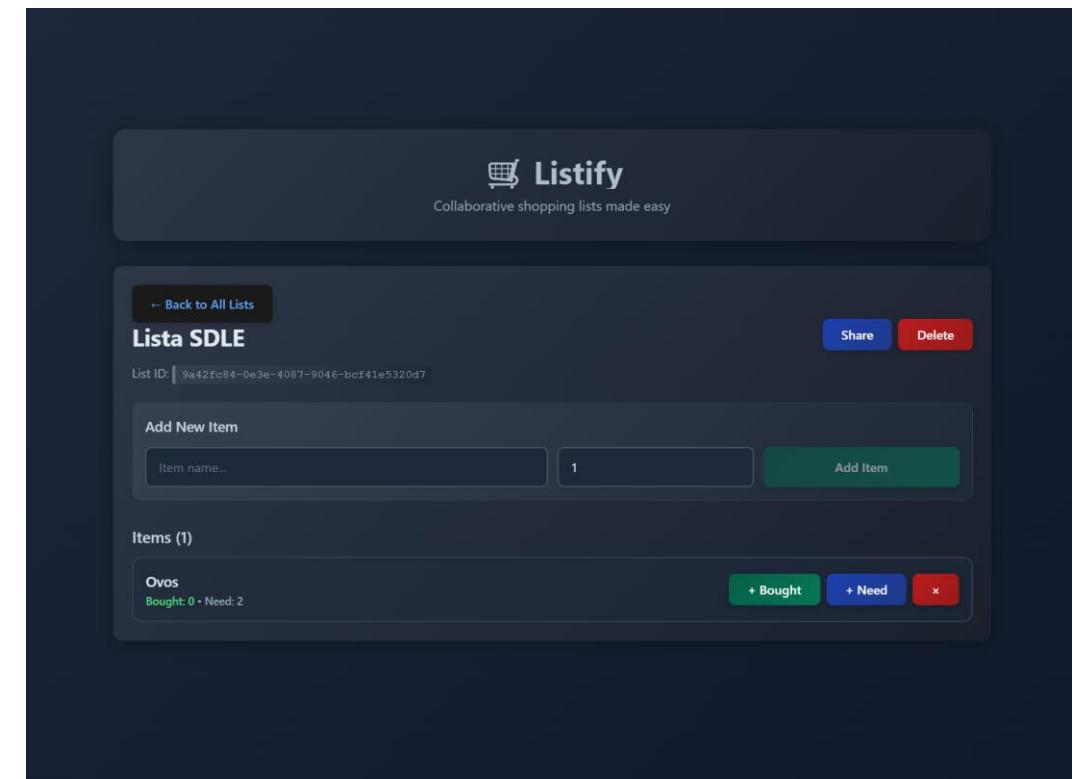


User Interface

Main page:



List page:



Main Challenges

- **CRDT Implementation:** The implementation presented several issues, particularly with merge operations that occasionally caused products to disappear.
- **Client Logic:** Managing local copies of CRDTs on the client side led to inconsistencies, with incorrect or outdated values sometimes being displayed in the user interface.
- **Server Logic:** Ensuring reliable storage and accurate replication of lists was one of the most significant challenges.
- **Client–Server Communication:** Transmitting CRDTs required sufficient contextual information to be correctly interpreted by the receiving side. Additionally, the communication framework had to be migrated from ZeroMQ to WebSockets, as WebSockets provided greater control and flexibility compared to the patterns supported by ZeroMQ.

Conclusions

- This project successfully delivered a **shopping list application** that combines **local-first functionality** with **cloud-backed storage**, enabling seamless list sharing among users.
- Through its development, we were able to **demonstrate the practical application of CRDTs** in resolving synchronization challenges and explore how a **local-first approach** can be effectively integrated into real-world systems. The outcome highlights both the **feasibility and advantages** of building resilient, collaborative applications that scale while maintaining data consistency and user autonomy.