Shopping List on the Cloud

Introduction

This project aims to develop a local-first shopping list application, emphasizing a user-centric approach. On one hand, it grants users the ability to store and access information directly on their devices, ensuring data persistence and accessibility offline. On the other hand, a cloud-based infrastructure to enhance collaboration. This multifaceted approach aims to optimize user experience by combining the benefits of local data management with the flexibility and scalability of cloud-based services.

Application Overview

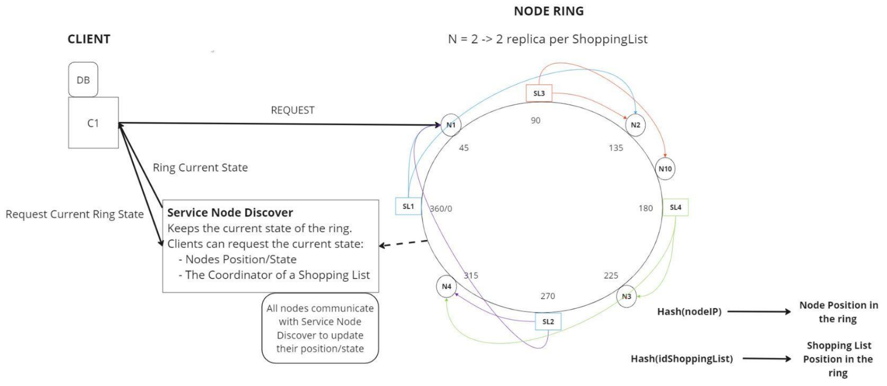
The Local-First Shopping List Application offers a seamless and collaborative way for users to create, update, and delete shopping lists. It prioritizes user control and data persistence, enabling local management on devices and cloud-based synchronization for sharing lists across multiple devices. Users can interact with the app using specified commands:

- Get {Shopping List Id}
- Pull {Shopping List Id}
- Push {Shopping List Id}
- Create {Shopping List Id}
- Add {Shopping List id} {Item Name} {Item Quantity}
- Remove {Shopping List id} {Item Name} {Item Quantity}

Users have their local versions of shopping lists, and they can create, update, and delete items on these lists. Much like Git, users can push their alterations to the cloud, ensuring that their changes are safely stored and synchronized across multiple devices through the *push* command. With the *pull* command, users can pull the most updated version of their shopping list from the cloud to their local device, ensuring that they always have the latest and most accurate information. For quick and easy reach to their shopping lists, users can utilize the *get* command to locally access their lists. This command ensures that users can always view their lists, even in offline mode. Using the *add* and *remove* commands, users can add or remove items from their shopping list, respectively.

Architecture Overview

This architecture comprises three essential components: the client component, the ring component, and the node service discovery component.


CLIENT
DB
C1
REQUEST
Ring Current State
Request Current Ring State
Service Node Discover
Keeps the current state of the ring.
Clients can request the current state:
- Nodes Position/State
- The Coordinator of a Shopping List
All nodes communicate with Service Node Discover to update their position/state
NODE RING
N = 2 -> 2 replica per ShoppingList
SL3
SL1
SL4
SL2
N1
N2
N10
N3
N4
90
45
135
360/0
180
315
225
270
Hash(nodeIP) → Node Position in the ring
Hash(idShoppingList) → Shopping List Position in the ring

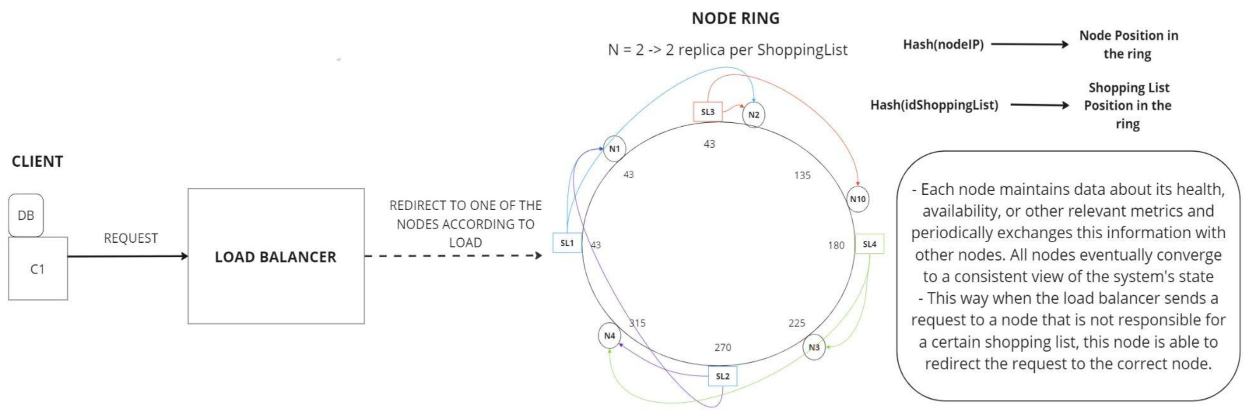
Client Component: This component manages the creation, modification, and deletion of shopping lists on the local device.

Node Service Discover Component: The node service discovery component serves as a central service that maintains awareness of the entire system's state. Each node in the ring periodically reports its state and health to the node service discovery component. Clients can request the ring state to discover which node is responsible for handling their requests.

Ring Component: This component is designed using consistent hashing to distribute the load across multiple nodes. Each node is assigned a unique position in the ring by applying a hash function to the node's IP address. The shopping list IDs are also hashed to be assigned a position in the ring. Each shopping list has a coordinator that corresponds to the closest node in a clockwise direction. The coordinator is responsible for replicating the shopping list to all N - 1 successors. This ensures that the data is replicated in multiple nodes allowing users to access their lists even if a node becomes unavailable, which guarantees availability. Moreover, consistency is maintained by propragating changes to shopping lists to all replicas during writes, ensuring that all copies remain synchronized and up to date.

Other approach

Although the architecture explained was the primary solution developed by our group, we also concluded that this approach has a drawback: the Node Service Discover component is a single point of failure. In other words, if this component crashes or fails to respond for any reason, our entire system will enter failure since clients cannot know which node to send their requests. With this concern in mind, we developed an alternative solution for discussion that still uses a ring-based structure, but this time the nodes exchange their state information using a gossip-based protocol in which nodes periodically exchange state information about themselves and other nodes they know about.


NODE RING
N = 2 -> 2 replica per ShoppingList
Hash(nodeIP)
Node Position in the ring
Hash(idShoppingList)
Shopping List Position in the ring
CLIENT
DB
C1
REQUEST
LOAD BALANCER
REDIRECT TO ONE OF THE NODES ACCORDING TO LOAD
N1
N2
N10
N3
N4
SL1
SL2
SL3
SL4
43
43
43
135
180
225
270
315

- Each node maintains data about its health, availability, or other relevant metrics and periodically exchanges this information with other nodes. All nodes eventually converge to a consistent view of the system's state
- This way when the load balancer sends a request to a node that is not responsible for a certain shopping list, this node is able to redirect the request to the correct node.

Shopping List CRDT

For the CRDT we thought of using an aworset (Add-Wins Observed-Remove Set) where elements are pairs consisting of a string (representing an item on the shopping list) and a pncounter (to track the quantity of each item). The PNCounter allows increments and decrements, by keeping a separate account of increments and decrements. An Add Wins Observed Remove Set is a set that allows element additions and removals. The removals only affect the elements that are visible locally, so that a concurrent removal and addition of the same element will result in the element still being present after joining the sets.