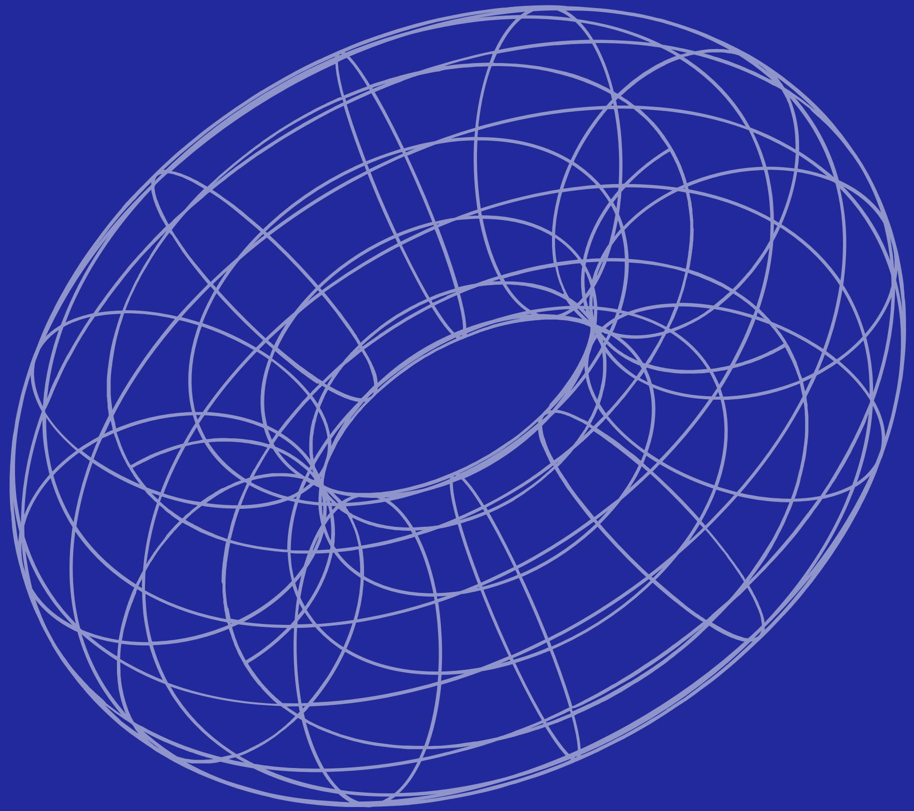
U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Shopping Lists on the Cloud

Large Scale Distributed Systems
SyncShopSquad

Rui Pires	up202008252
João Reis	up202007227
João Teixeira	up202005437
Francisco Serra	up202007723

Requirements

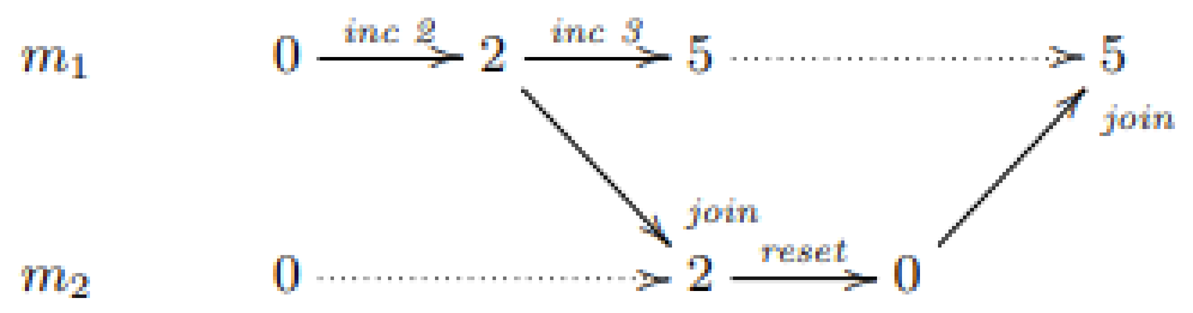


- The goal was to develop a local-first shopping list app allowing users to create and modify lists locally, while also enabling the possibility of sharing lists via the cloud.
- Due to concurrent changes in shopping lists represented by the same ID, it was highly desirable to pursue a high availability strategy.
- Finally, it was also suggested for this system to follow the Amazon Dynamo architecture, in order to provide a scalable solution that allows millions of users.

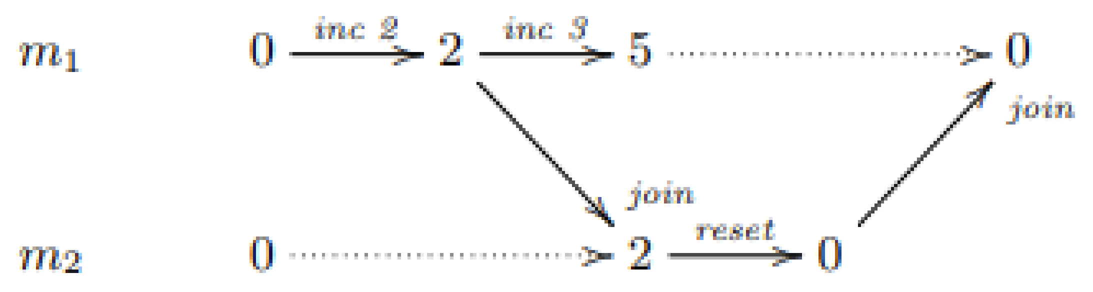
Technical Solution

High availability Strategy (CRDTs)

In our exploration of traditional counter CRDTs for the Cloud Shopping List Project, we faced the 'observed-reset' anomaly, where a counter reset doesn't fully reverse all observed operations in merge scenarios, particularly with concurrent updates from other replicas. This led us to the solution presented in the paper 'The problem with embedded CRDT counters and a solution,' which advocates for a 'remove wins' counter design. This approach effectively addresses the anomaly and every increase (and decrease) observed in a replica is accurately reset when an entry is removed. Furthermore, any simultaneous operations are influenced by the reset, leading to a 'remove wins' scenario. We made some adjustments to the CRDT outlined in the paper, as certain fundamental operations were tailored only for an 'add wins' counter, which was not required for our purposes.



observed-reset anomaly



remove-wins counter design

Technical Solution

High availability Strategy (CRDTs)

State Composition:

- **Dot Store:** Maps replica IDs to a pair of integers (positive and negative counts).
- **Casual Context:** Essentially a version vector

Core Operations:

- **Initialization:** Starts with empty dot store and casaul context.
- **Increment/Decrement:** Alters the counter by finding/creating and active entry in the dot store.
- **Reset:** Clears the dot store, while keeping the casual context.
- **Value:** Calculates the counter's value by summing positive counts and subtracting negative counts.

Join Operation:

- Merges two counter by combining their values and maintaining consistency and causality.

$$\begin{aligned}(m, c) \sqcup (m', c') \quad = \quad & (\{d \mapsto m(d) \sqcup m'(d) \mid d \in \mathsf{dom}\, m' \cap \mathsf{dom}\, m\} \cup \\ & \{((j, n), v) \in m \mid n > c'(j)\} \cup \{((j, n), v) \in m' \mid n > c(j)\}, \\ & c \sqcup c')\end{aligned}$$

Technical Solution

Client

Clients have the following information:

- **email:** the client's email;
- **loadBalancerIP:** the address where the load balancer is running;

The **client** can have two types of interactions:

- **Online**: Push/pull to/from the servers to update shopping local lists
- **Offline**: Create a list, edit/show an existing list.

Online - First the client needs to insert the email from the list:

- Push: We load the contents from the list to a writer (mime object) that will be sent via a POST request to the load balancer.
- Pull: First we make a GET request to the load balancer to get the file. Then we create a new list with the one we received. If we already had a local copy of that list we merge the new old one with the new one. Otherwise we create a new empty list and join it with the one we received. After this we save the merged list to a local file.

Both pull and push interactions have a defined number of maximum retries of 3 and the interval between them of 2 seconds if there's any connection issues.

Technical Solution

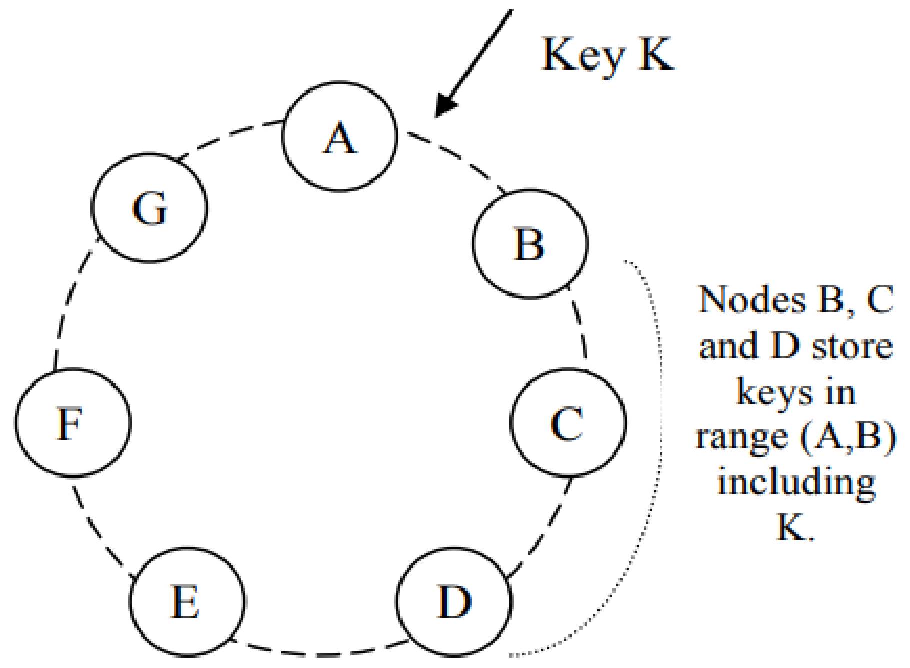
Client

Offline:

- Create a new list: The client can choose whether the list has their email associated or a different one;
- Edit an existing list - After providing the email the client can choose 3 options:
  - Add item - After inserting the new item and the quantity we check if the item already exists in the list. If it exists we only increment the quantity otherwise we add the new item to the list and increment the quantity;
  - Remove item - After inserting the item to remove we delete that item from the list;
  - Edit item quantity - First the client inserts the item name then it chooses whether they want to increment or decrement. (If the item doesn't exist then we first create it and then execute the action) :
    - Increment: Client chooses the quantity by which they want to increment the article;
    - Decrement: Client chooses the quantity by which they want to decrement the article.
- Show an existing list - An email is required to show the list, even if it’s from the client.

Technical Solution

Load Balancer



- This is the server responsible for the even distribution of data across all servers, achieved by the use of a consistent hashing algorithm.
- It also manages all server connections by adding nodes and, consequentially adding their representative virtual nodes to the ring.
- Functions as a proxy server between the client's requests and the server nodes, routing each request to the correct server in the hash ring, because this server is the only one with full knowledge of the current state of the ring.
- Whenever a client attempts to put (push) a new shopping list in the cloud, the Load Balancer, not only links the list to the right server node in the ring, via consistent hashing but also attempts to replicate the list on a predefined number of nodes that are situated ahead of the selected node (see image on the left).
- In our system, we used the following constants:
  - Replication factor = 3
  - Number Of Virtual Nodes = 4
  - Hashing Algorithm: SHA-256
- Lastly, it is also worth mentioning that the Load Balancer provides some degree of information about the hash ring to all the node servers whenever a new server joins the ring. The content of this information will be discussed ahead.

Technical Solution

Cloud Servers

The **servers** have the following information:

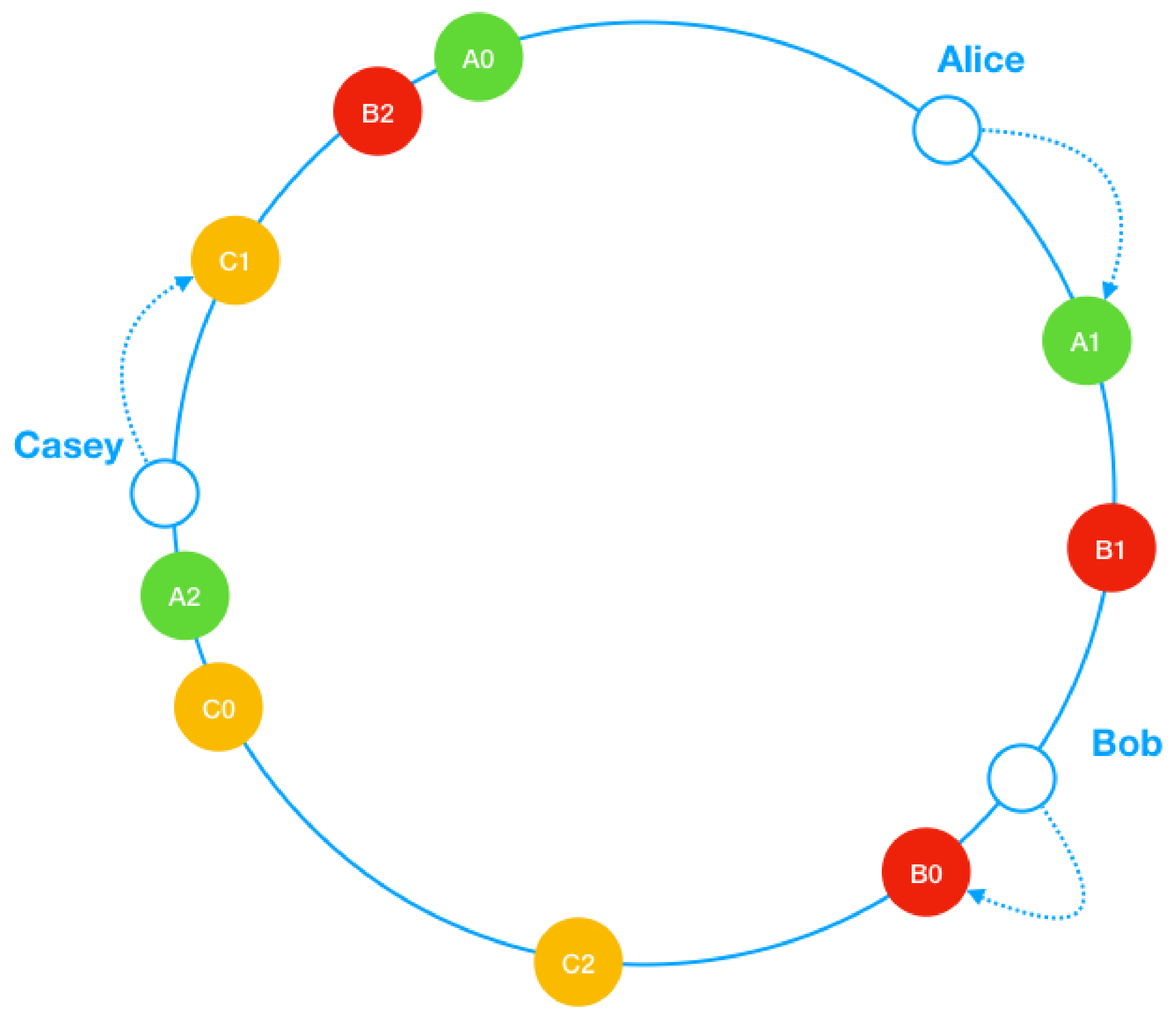
- **port:** the port in which they are running;
- **name:** the server's name;
- **loadBalancerIP:** the address where the load balancer is running;
- **db:** the server's database where its shopping lists are stored;
- **nodes:** contains important information regarding all the servers nodes in the hash ring.

They are able to **communicate** with:

- **Load Balancer:** the servers communicate with the load balancer to receive the **push** and **pull** requests sent by the clients:
  - In case of a **push** request, we first check if there already exists a shopping list with the same name as the one sent in the request. If it does, then they are merged using the **CRDTs**. If not, then a new entry is added to the database with the information from the shopping list received;
  - In case of a **pull** request, if a shopping list exists with the requested name, then we simply send its contents to the **Load Balancer**, which will then send it to the client. If not, an **error** is sent.
- **Other servers**: The servers are able to communicate between them. This is needed for the synchronization process, as well as the process of adding a node which will be explained shortly ahead.

Technical Solution

Adding a Server to the Cloud



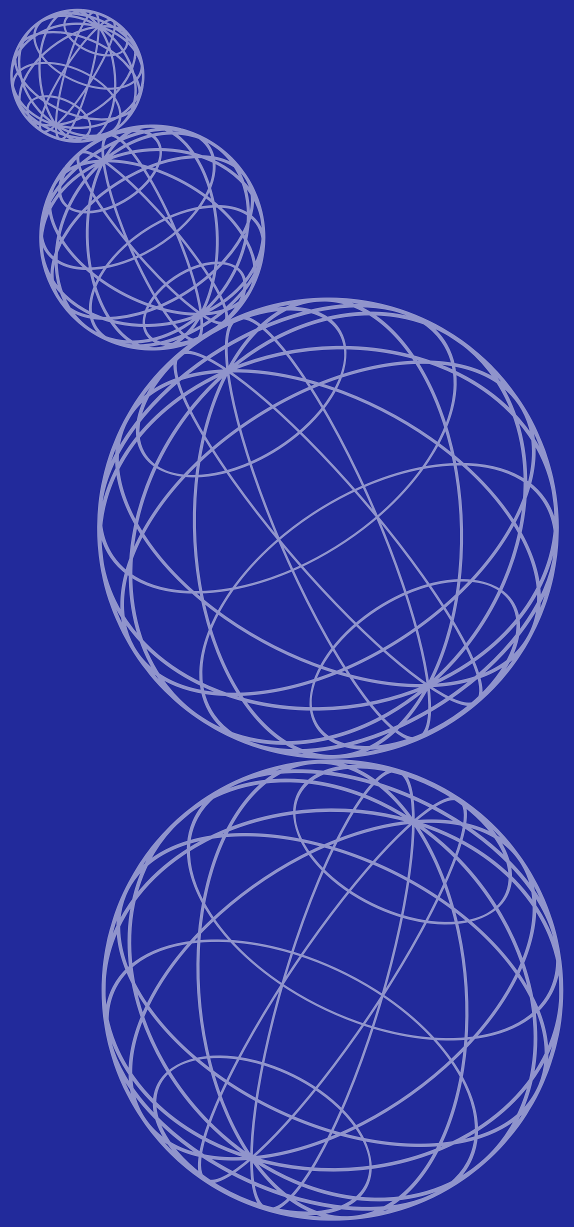
- Whenever a server node is added to the system, complications arise. There will be a need for key re-allocation between servers, which proves to be a complex runtime operation.
- To solve this, as mentioned before, our servers are able to store some degree of information about the state of the hash ring, namely:
    - The server knows all the hashIDs of its own nodes (real or virtual) in the ring.
    - For each node of the server (real or virtual) present in the ring, the server knows the hashID of their next 2 front node neighbors as well of their 2 back node neighbors.
- With just this knowledge each server is capable of knowing all the hash ranges in the ring for which it is responsible.
- This information is updated by the load balancer, whenever a new server is added to the ring.
- Lastly, the key re-allocation is achieved by the newly added server which communicates with its first front node neighbor, requesting all the keys that are situated between itself and its first back node neighbor.

Technical Solution

Synchronization Process

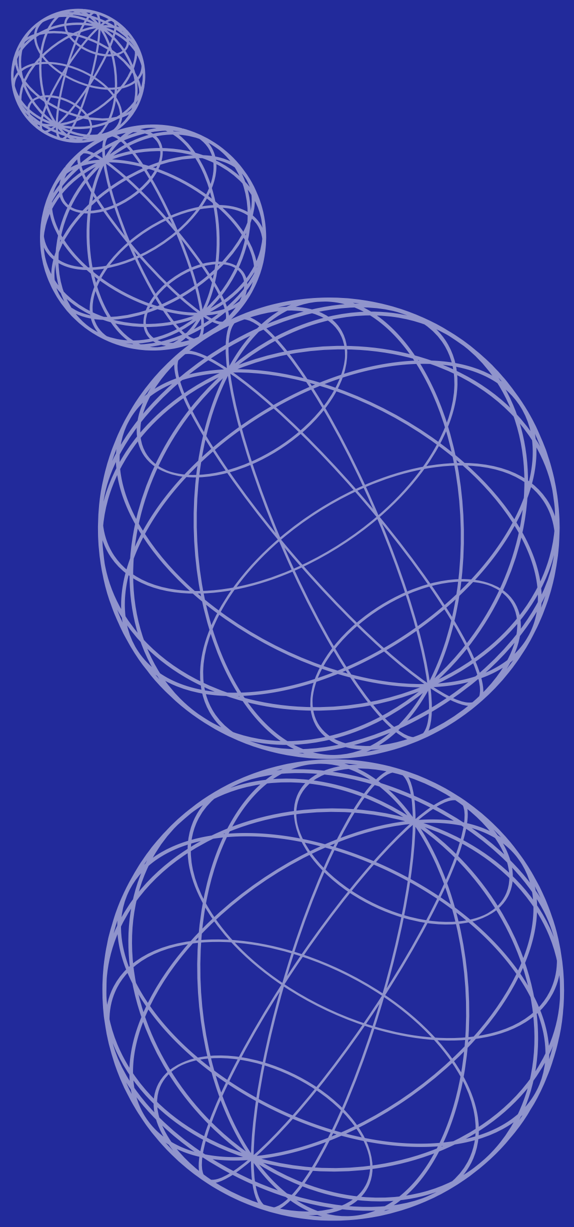
The server systematically processes each node, calculating the hash space between the node and its immediate back neighbor. It then fetches the CRDTs (Conflict-free Replicated Data Types) from the database located within this hash space and dispatches them to the replicas via a post request. When the replicas receive this post request, they consult their databases to find and retrieve CRDTs within the specified hash space, which are then sent back to the original server as a response. The replicas then examine the CRDTs received; if the CRDT already exists in their database, they merge (join) it with the existing one and update the database with the merged result. If the CRDT is new, they add it to their database.
Subsequently, upon receiving these responses, the server updates its database accordingly. For each incoming CRDT, if it is already present in the database, the server merges it with the existing one. If the CRDT is new, the server incorporates it into the database. This process ensures synchronized and updated data across all nodes and replicas in the network.

Limitations



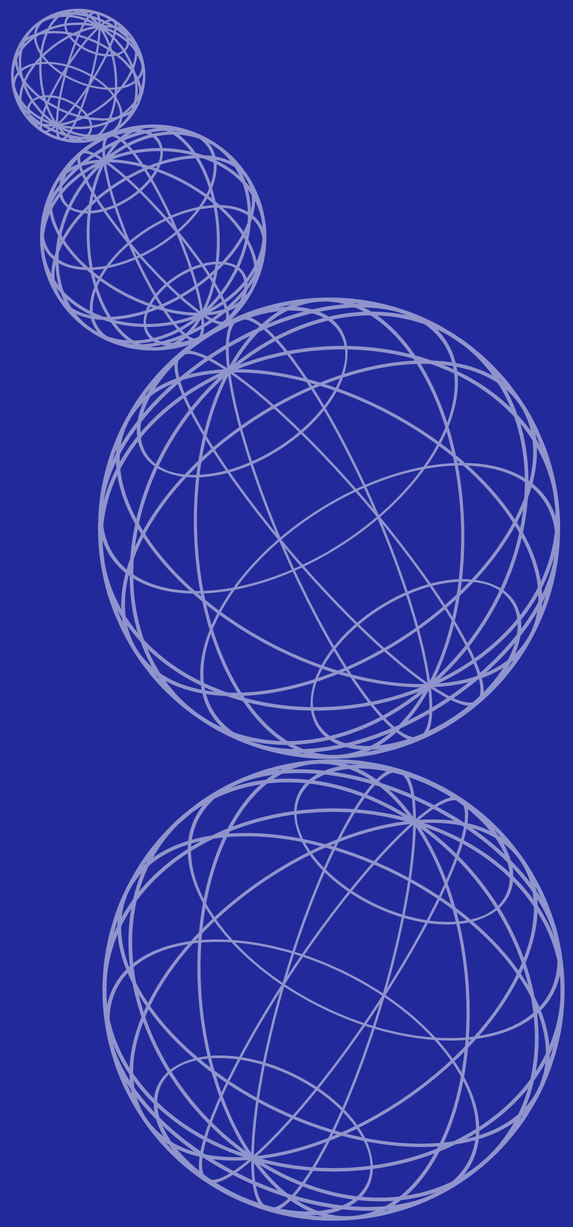
- The main limitation of our solution is the fact that the load balancer server induces a Single Point of Failure (SPoF) in our system. This exists because the whole system would not function in the event of a failure in the load balancer server, which is responsible for managing the clients' requests as well as operations that alter the state of the hash ring.
- We also did not implement the removal of a node from the ring. The complexity of this operation proved to be significantly high, and we decided not to implement it. Instead, whenever a server stops working, we simply wait for it to come back online so that the synchronization process can update all of its outdated shopping lists.
- Lastly, and similarly to the above limitation, whenever a new node is added to the ring, we are not proceeding to remove them from servers that used to replicate them and no longer need to. They become “dangling lists”.

Future Work



- Creating a UI for better experience and simpler use.
- Eliminating the SoPF in the load balancer, which could be done via the use of epidemic algorithms. That way every server node would know the state of the hash ring which would allow for failures in the load balancer because a new node would take its place.
- Implementing an admin console, for the operations of adding and deleting servers from the system.
- Deploying the system so that it could be used by anyone.

References



Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. 2016. The Problem with Embedded CRDT Counters and a Solution. https://repositorium.sdum.uminho.pt/bitstream/1822/51503/1/Problem-Solution-Counters-PAPOC2016.pdf

DeCandia G., Hastorun D., Jampani M., Kakulapati G., Lakshman A., Pilchin A., Sivasubramanian S., Vosshall P., and Vogels W.. 2007. Dynamo: Amazon’s highly available key-value store. https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

Thank you all for your attention.
Sincerily, SyncShopSquad.