FEUP - MEIC

Large Scale Distributed Systems

SHOPPING LIST APP

- Carolina Gonçalves - up202108781
- João Costa - up202108714
- Marco Costa - up202108821
- Rafael Teixeira - up202108831

PROBLEM/REQUIREMENTS DEFINITION

Local-First Shopping list app

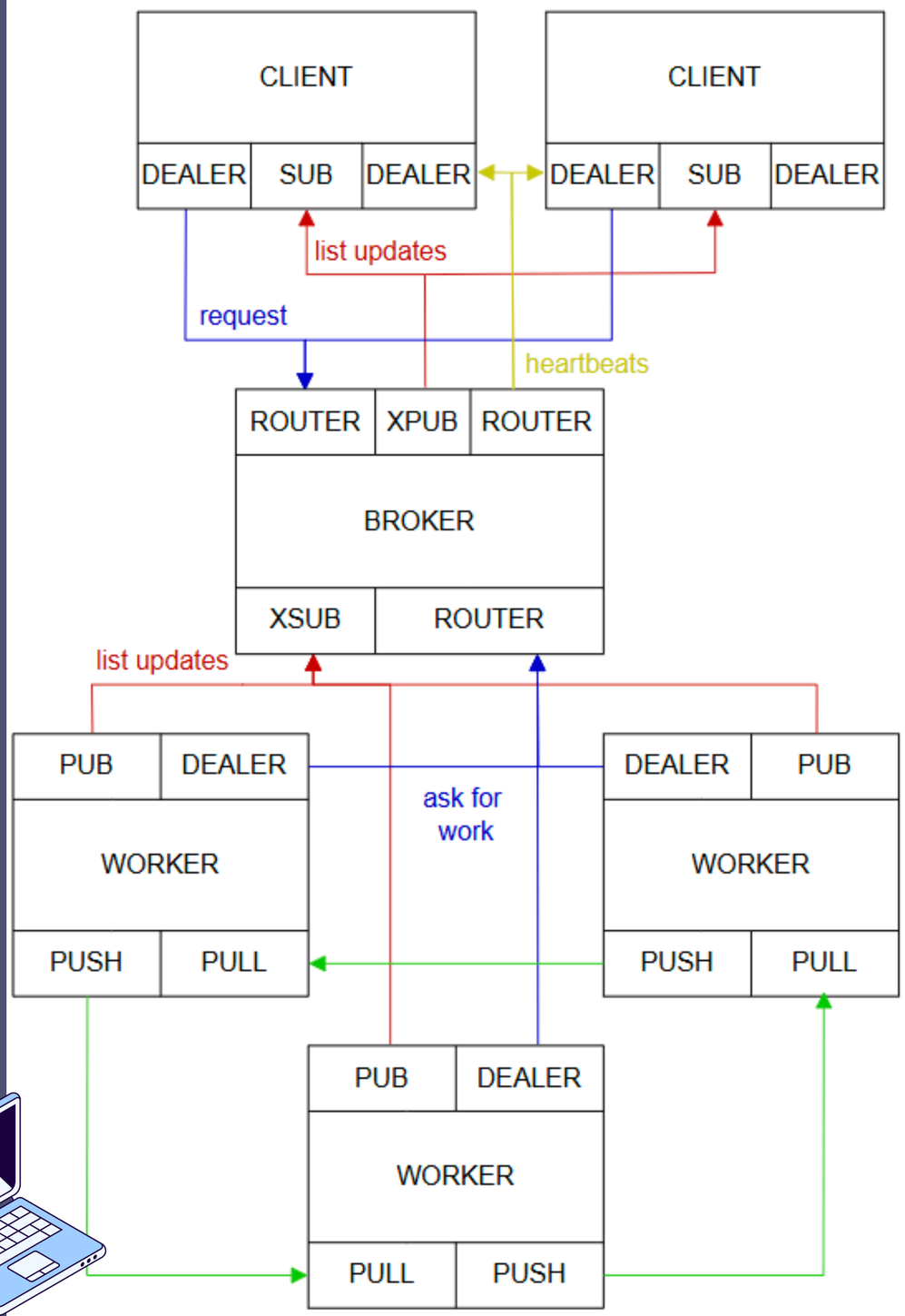
Create List, Get List, Update List, Delete List

High Availability

ZMQ

Initially we had difficulties defining a communication structure, which led us to go through a phase where we were constantly changing it until we got one we liked. This was the final structure considered:

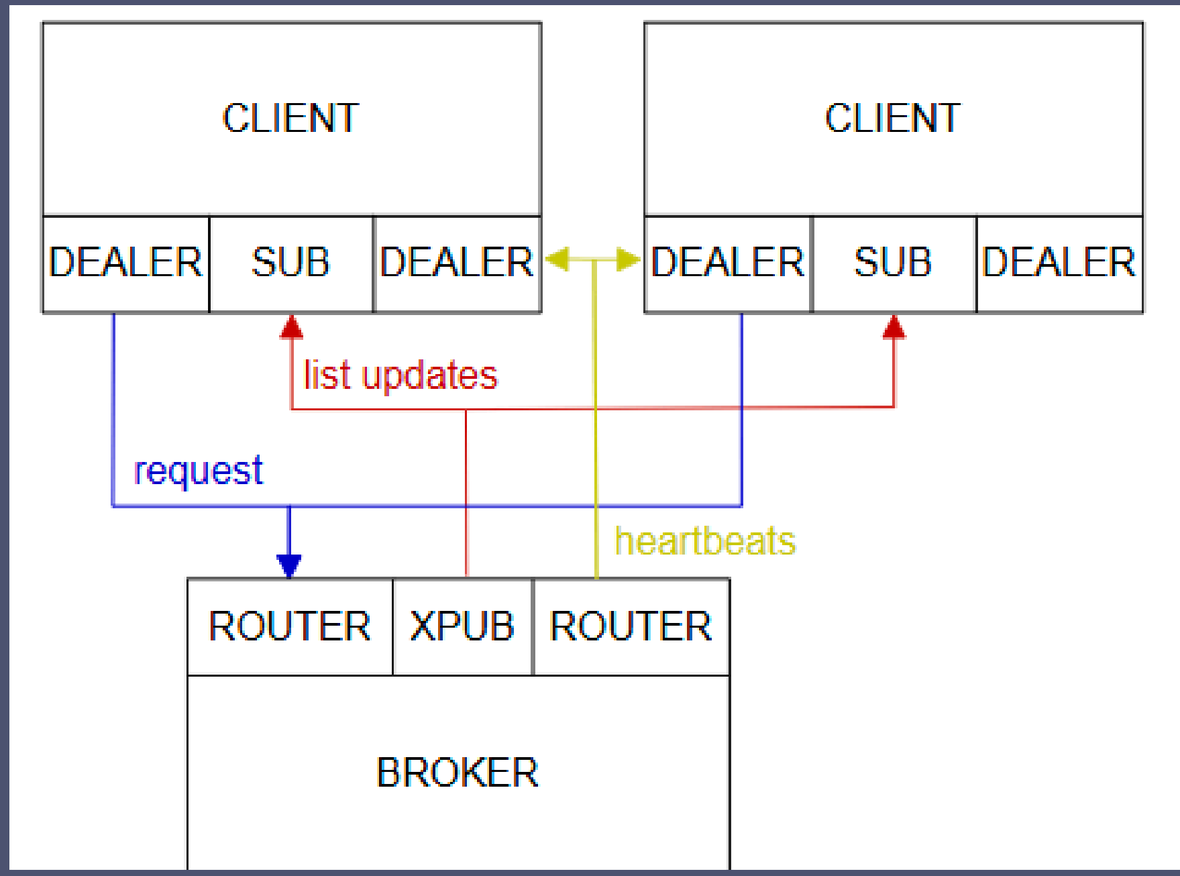
Our objective was to create an **asynchronous system** to provide a better user experience to the final user and to challenge us on making a more complex project.



ZMQ - CLIENT

DEALER:

- Submit requests to the broker;
- Allows clients to send multiple asynchronous requests **without waiting for a reply**;
- This is essential for a **non-blocking**, **responsive UI**, since users can continue to use the app (like editing lists) without being "stuck" waiting for a server response;



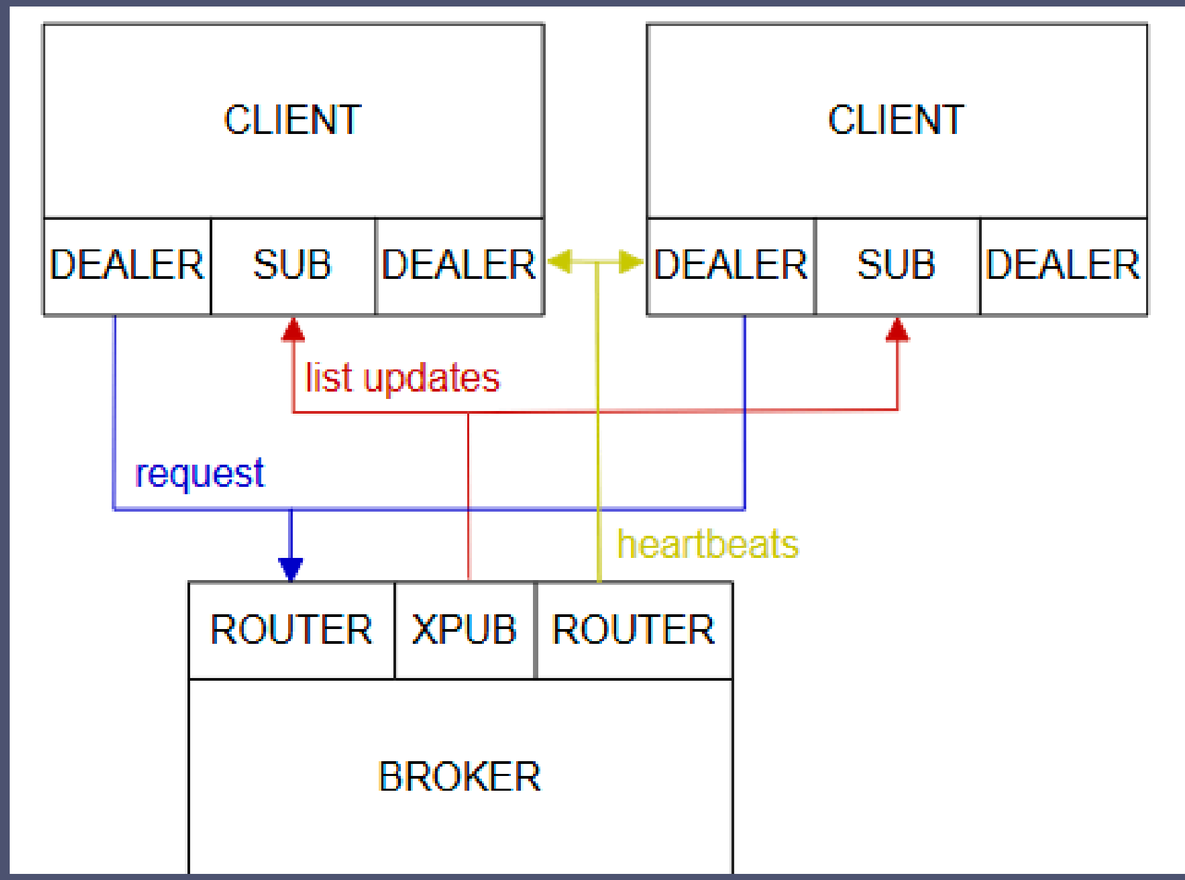
Alternative option?

- A **REQ** (request) socket would require the client to block and wait for a reply before sending the next request. This would harm user experience;
- A **REQ** could only handle 1 request at a time;
- A **PUSH** socket would not expect a response;

ZMQ - CLIENT

SUB:

- Receive updates from the lists that a client is subscribed to from the broker;
- Allows clients to subscribe to **specific updates** (list they only have access to), which **reduces unnecessary data transfer**;
- Allows for **real time notifications**;



Alternative option?

- A **PULL** socket would force clients to poll for updates, which is inefficient for real-time updates;
- A **PULL** socket would require clients to process all updates since it cannot filter messages;
- Using a **DEALER/REQ** would introduce delays, as clients would need to check for updates periodically;

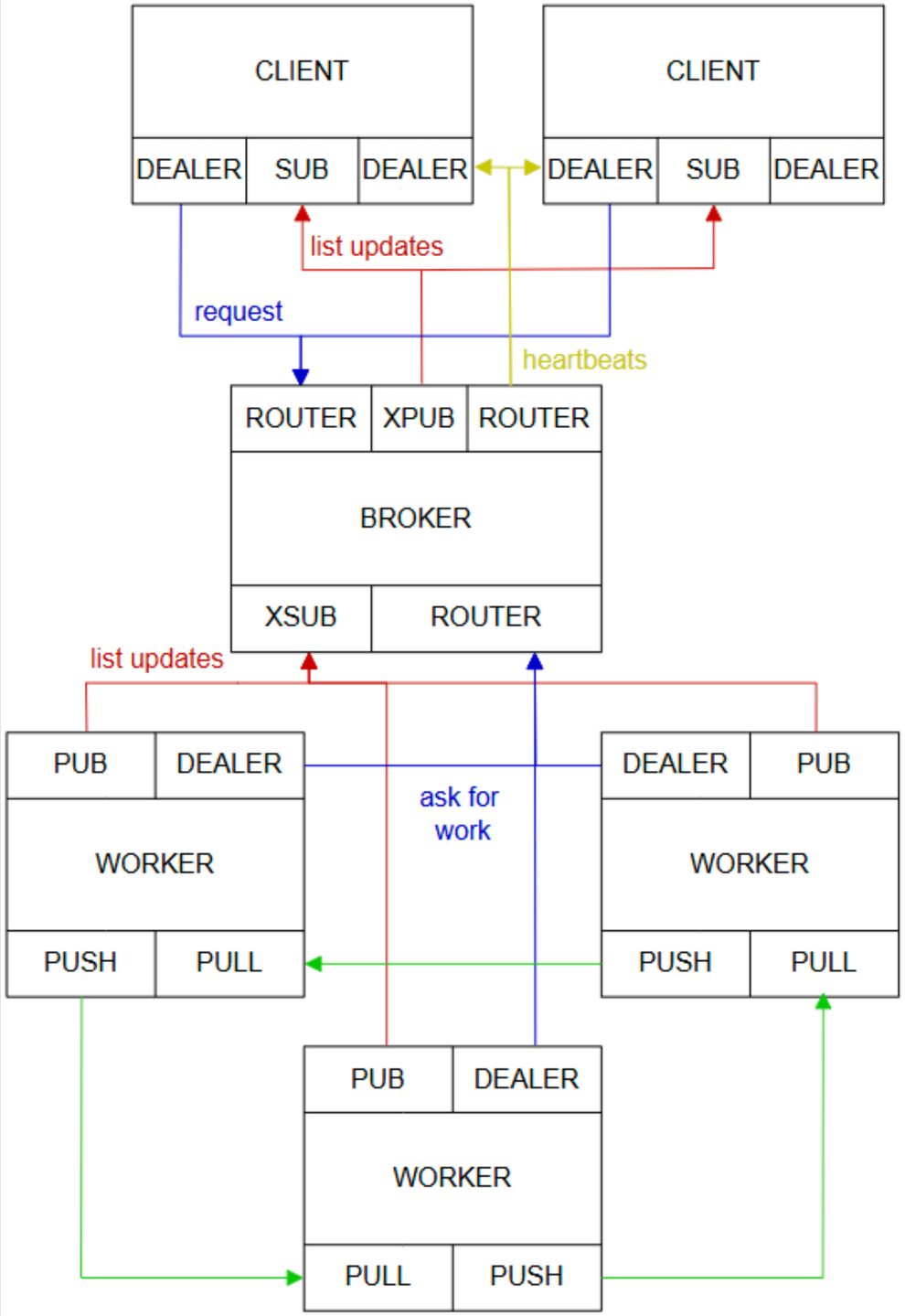
ZMQ - BROKER

ROUTER:

- **Redirect requests** received from the client to the workers; Send **HEARTBEATS** to the clients when atleast one worker is active;
- Allows the broker to receive **multiple concurrent requests** from multiple clients and queue them for processing;
- Can send requests to **any** of the connected workers;
- Tracks the **identity of the client** that made the request, which allows the broker to know which client to respond to after the worker completes the task.

Why is ROUTER better than REP?

- REP only supports one connection at a time;
- REP cannot send a request to a worker that didn't originally send a message;



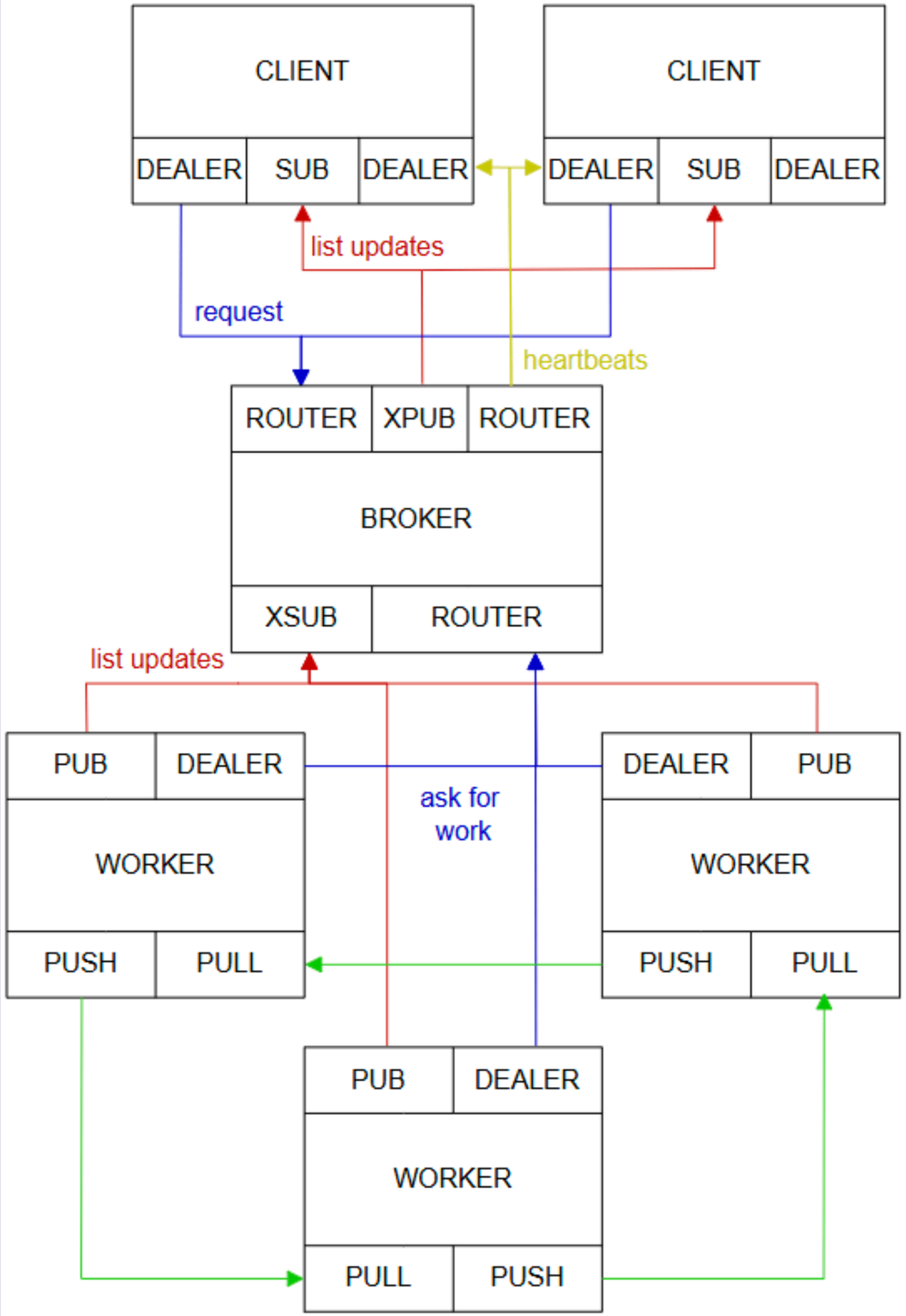
ZMQ - BROKER

XSUB:

- **Subscribe** to **all the list updates** made from the workers when processing a client request;
- If a new worker connects, it can **automatically publish updates** to XSUB, and the broker will receive them;

Why is XSUB better than PULL?

- **PULL** socket would only receive updates from one worker at a time.



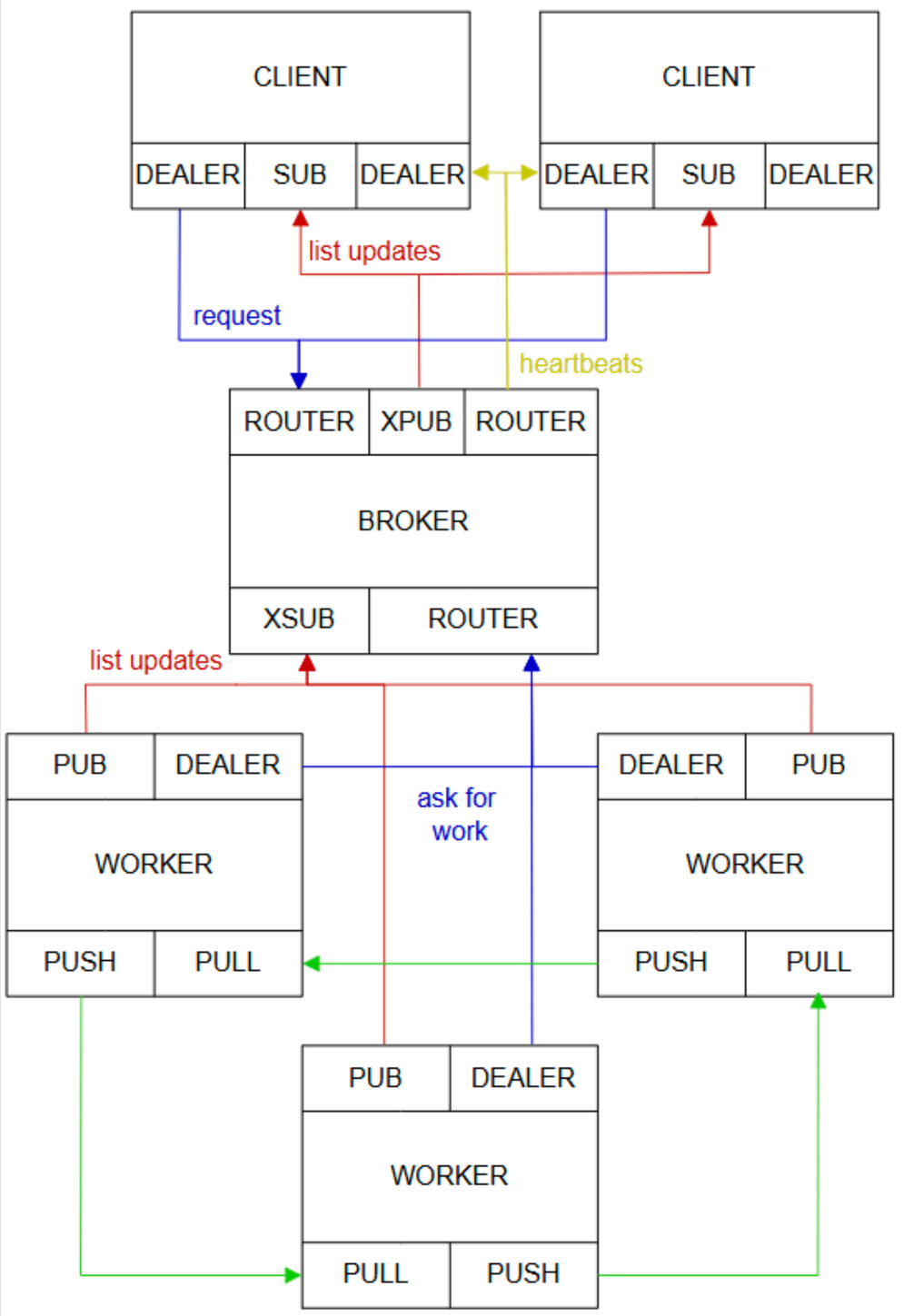
ZMQ - BROKER

XPUB:

- **Publish** to the clients the **list updates** made;
- Clients don't need to "ask" for updates, they receive updates **immediately when they happen**;
- Allows the broker to **manage multiple client subscriptions** for multiple lists;
- Dynamically supports **clients joining and leaving**;

Why is XPUB better than PUSH?

- **PUSH** socket would require clients to explicitly request data;
- **PUSH** doesn't have topic subscription, so it would send every update to everyone indiscriminately;
- **XPUB** allows clients to join and leave at any time, unlike **PUSH**.



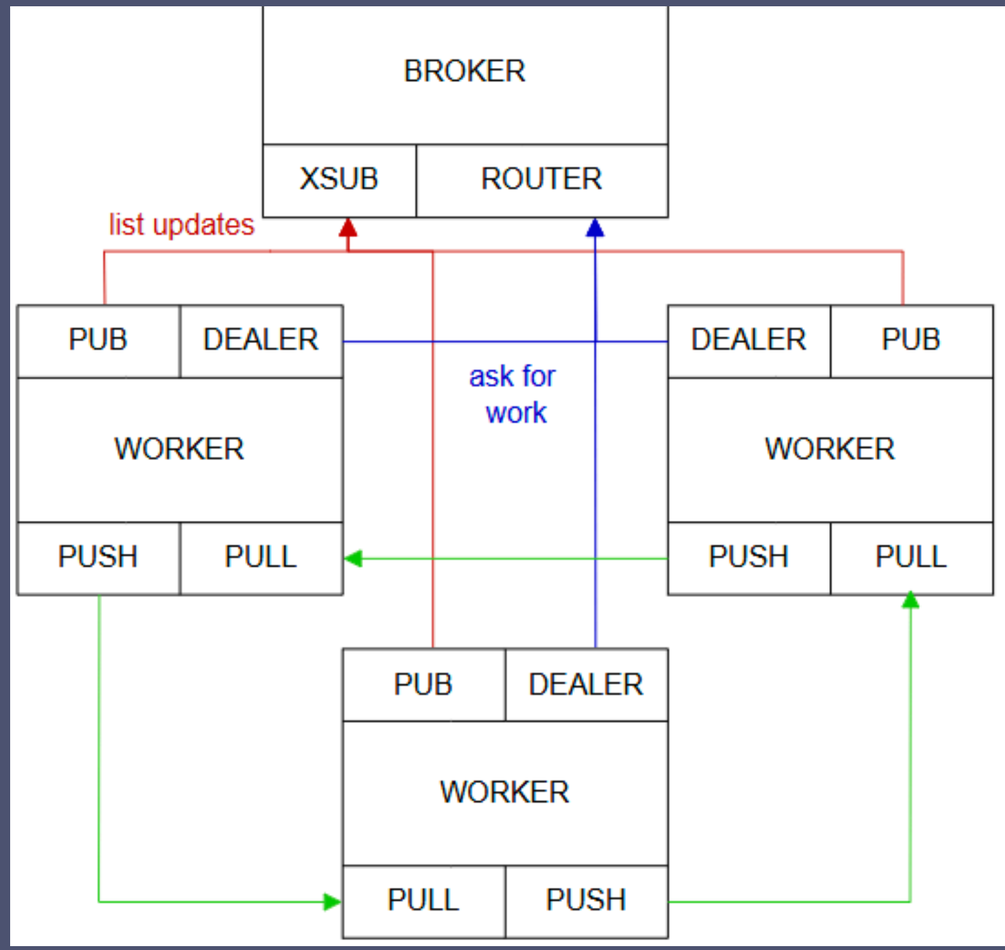
ZMQ - WORKER

DEALER:

- **Ask** the broker **for work** and **submit** the **reply of the client request** received;
- Allows workers to send **multiple replies concurrently without blocking** on each individual response;

Why is DEALER better?

- Unlike other sockets like **PUB**, which broadcast messages to all connected clients, **DEALER** ensures the **correct reply goes back to the broker**, maintaining proper routing.



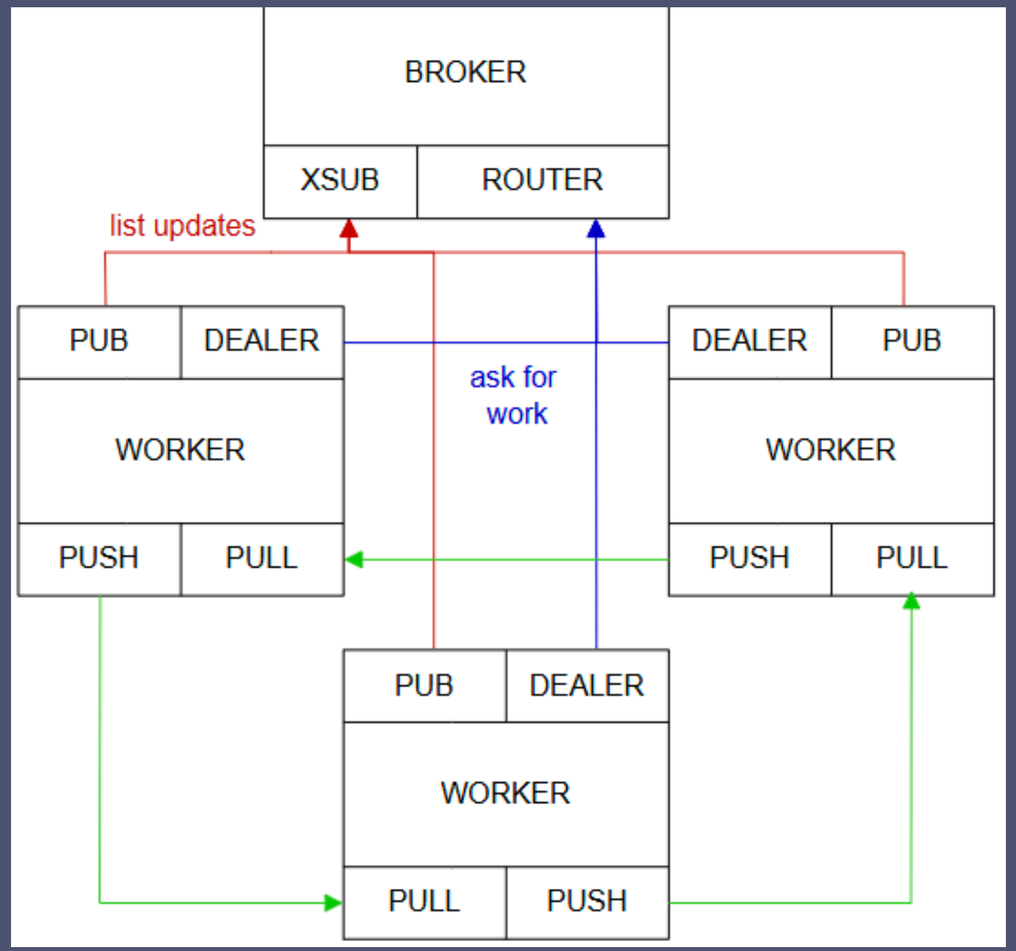
ZMQ - WORKER

PUB:

- **Publish** to the worker all the **updates made to lists**;
- Worker **publishes updates as soon** as they happen, ensuring that clients are kept in sync with the latest data without requiring clients to manually request updates;
- Allows for many clients to receive updates without increasing complexity;

Why is PUB better?

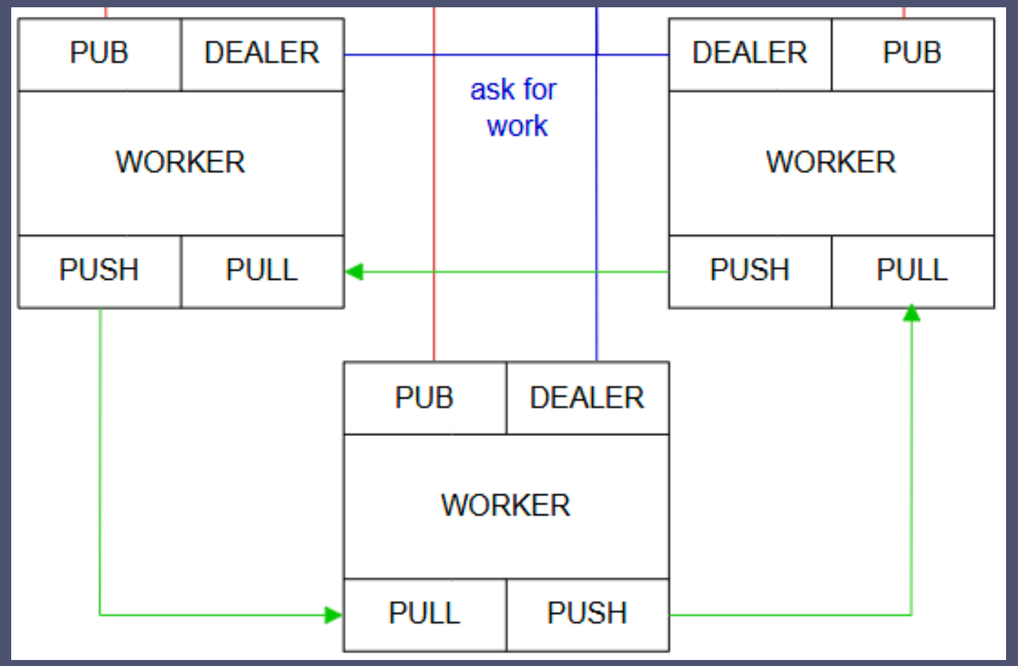
- Handles **large numbers** of subscribers efficiently, unlike other sockets that would require one-to-one communication (e.g., **PULL** or **DEALER**).



ZMQ - WORKER

PUSH/PULL:

- Establish worker's **connections** and **redistribute** tasks;
- Enables the worker to receive tasks in an **asynchronous**, **non-blocking manner** from other workers;
- Follows a no reply style, as no reply is expected on those actions.

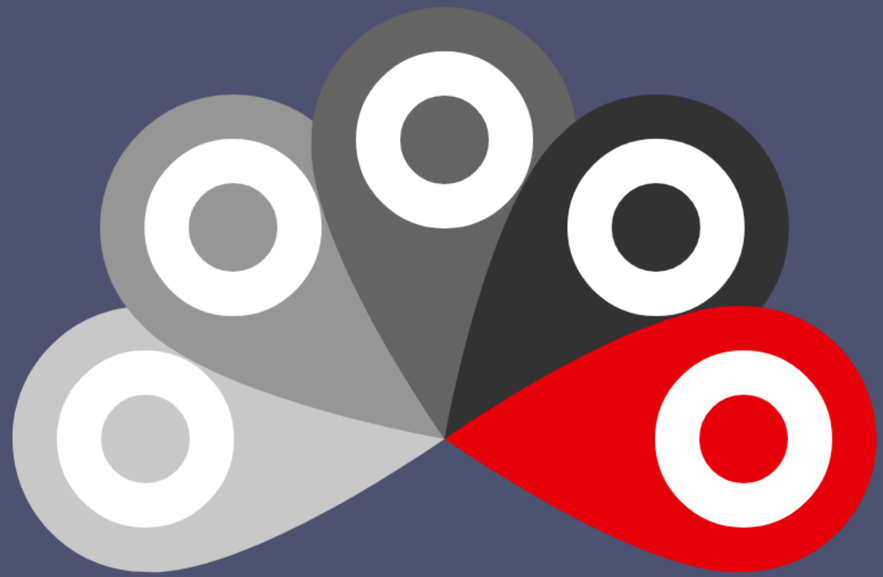


Why is PUSH/PULL better?

- Allows for **asynchronous**, **non-blocking** communication, which ensures that workers can perform tasks independently, **without waiting** for other workers to complete their work;

LOCAL FIRST

- The application works primarily locally.
- Firstly, it updates the customers' shopping lists in the respective JSON files associated with the same list.
- These files act as the data storage of our application, ensuring the integrity and accessibility of all modifications.
- Once at least one worker signals it is ready to process tasks by sending a "READY" message to the broker, the broker, in turn, notifies the client.
- This notification, implemented via "HEARTBEAT" messages, informs the client that the cloud-level (online) operations can commence. This architecture ensures a seamless transition from local data handling to distributed cloud-based processing, enhancing reliability and scalability.



CRDTS

In this project, data conflicts were inevitable. Implementing an appropriate CRDT that could represent the Shopping List was necessary in order to solve this problem.
The main requirements were:

- Allows for insertion and removal
- Allows for increments and decrements

Being efficient both in time and spatial complexity was also something we strived for.

The three options implemented:

GSet w/ PNCounter

AWORSet w/ PNCounter

ORMAP w/ CCounter

CRDTS

Requirements	INC	DEC	INS	REM	TE	SE
GSet w/ PNCounter	INC		INS	REM	TE	SE
AWORSet w/ PNCounter	INC	DEC	INS	REM	TE	SE
ORMAP w/ CCounter	INC	DEC	INS	REM	TE	SE

CRDTS - ORMAP W/ CCOUNTER

Advantages

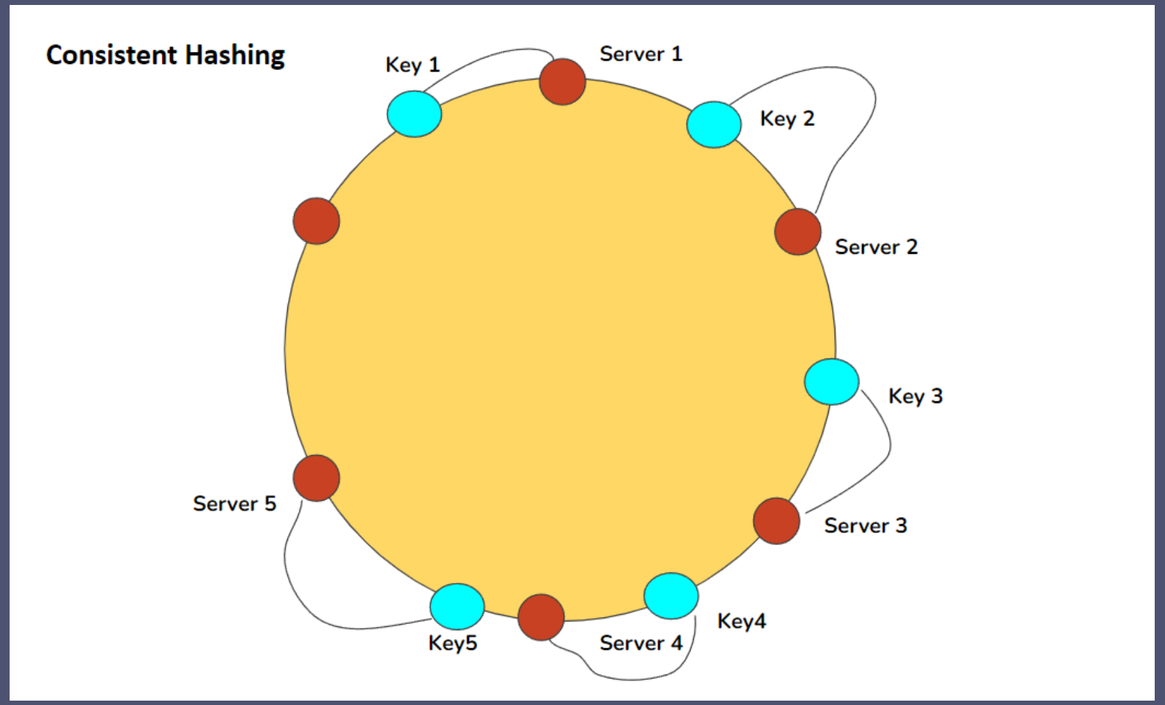
Allows for all operations natively;
Feels like we are working with a list of items;
It allows for easy compression and decompression;
It is implemented such that it can be easily extended for Delta CRDTs.

Disadvantages

It is using a little bit more space and more operations than needed as Delta CRDTs ended up not being implemented.

CLOUD COMPONENTS

A broker is in charge of overseeing the distribution of tasks in this system. The broker assigns a task to a worker who is available when it is received.
The task is then routed by the initial worker to the designated worker, who is identified by consistent hashing.
Even in situations when the workforce is dynamically changing, this method guarantees effective job distribution while preserving a consistent task distribution to employees.

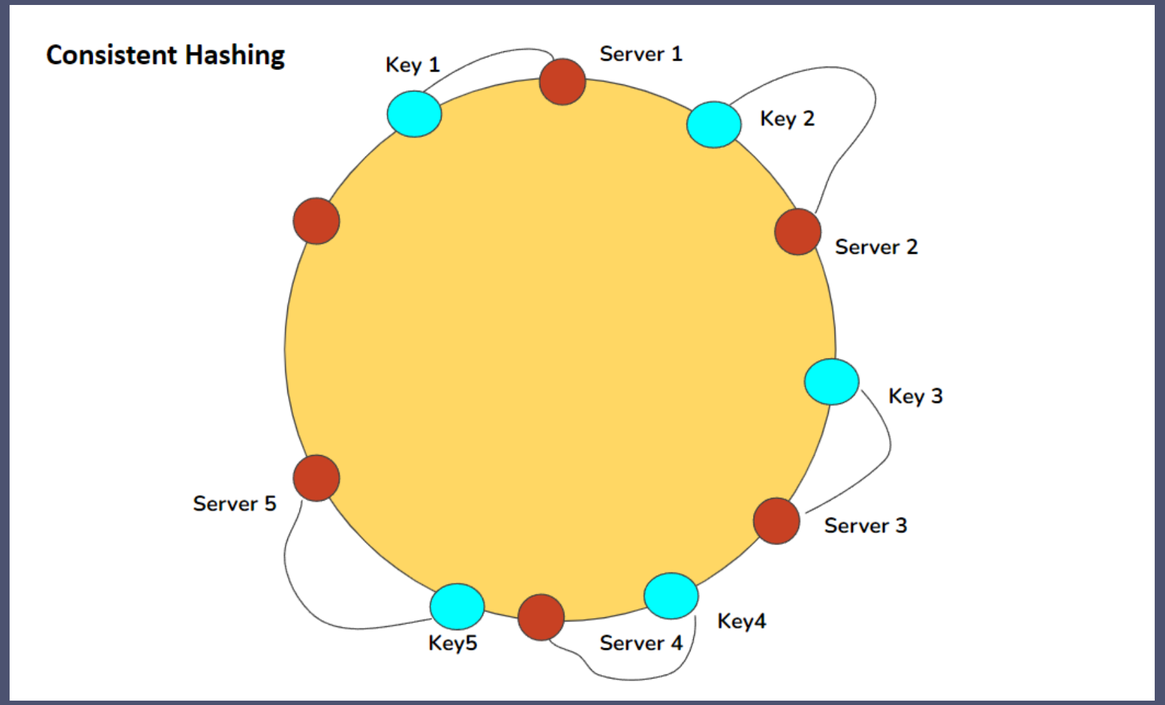


SHARDING

Each shopping list is identified by an **URL**. This **URL** is mapped into hash space and is used to identify which worker should take that request.
This means that each worker is also also mapped into the hash space, thanks to its **endpoint**.

This solution does not guarantee that the work is equally distributed, however, it is a step towards it. Furthermore, it improves the availability of our system.

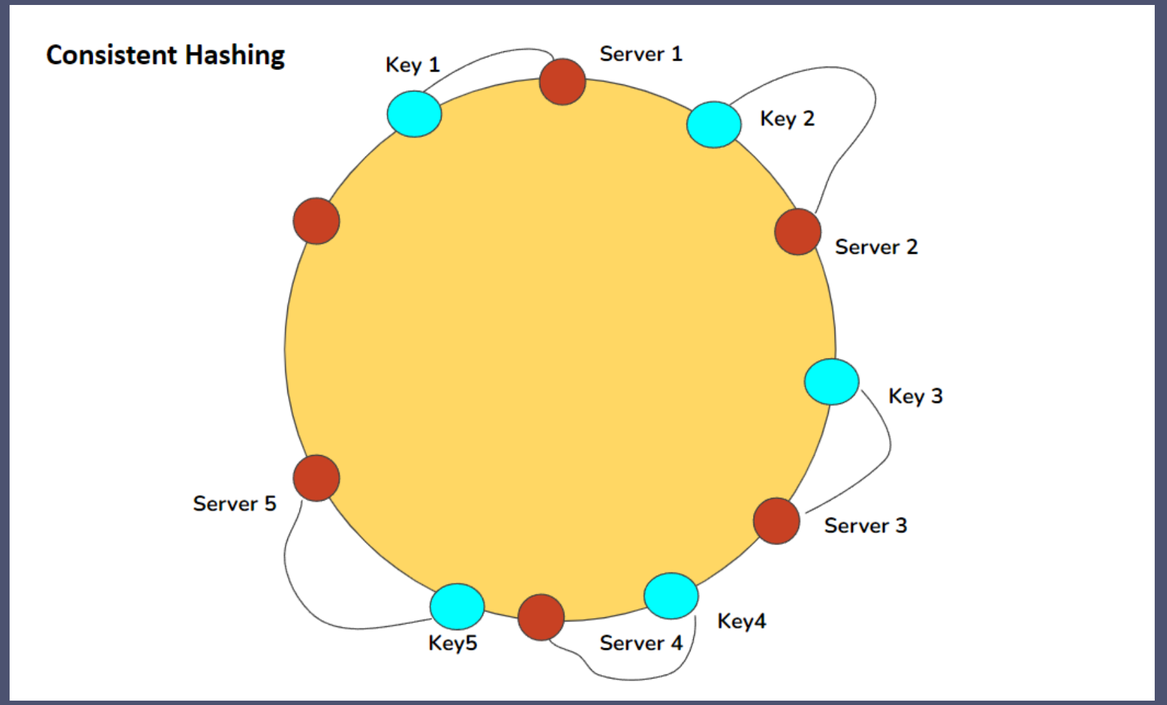
Ideally, each physical server would be represented in the ring as many **virtual workers**. This was implemented but not used as we couldn't deal with the intricacies related with replication



REPLICATION

In order to increase the availability of the system, when we are distributing the work to each worker, we replicate the task to the next **N** workers. This allows for the system to maintain service even if some workers go down.

This system is not prepared for dynamic insertions and removals of workers, as the system does not redistribute work to each of them on connect and disconnect.



SOLUTION LIMITATIONS

- When a new worker is added the work that was now supposed to be for it, is **not** redistributed.
- When a worker goes down its' work is **not** redistributed to other workers.
- The broker serves as a **single point of failure** in the system.
- We didn't implement **Virtual Nodes**.
- Our CRDTs are more complex than needed as they are **delta-based**. This could prove useful to reduce the bandwidth used during communication, but it is an inconvenience.

REFERENCES

- Decandia, Giuseppe, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. “Dynamo: Amazon’s Highly Available Key-Value Store.” https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf
- “ZeroMQ.” Zeromq.org. https://zeromq.org/
- Bieniusa, Annette, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. “An Optimized Conflict-free Replicated Set.” arXiv (Cornell University), January 1, 2012. https://doi.org/10.48550/arxiv.1210.3368
- Sypytkowski, Bartosz. “Optimizing State-based CRDTs (Part 2).” Bartosz Sypytkowski, May 9, 2021. https://www.bartoszsypytkowski.com/optimizing-state-based-crdts-part-2/
- “Local-first Software: You Own Your Data, in Spite of the Cloud,” April 1, 2019. https://www.inkandswitch.com/local-first/

REFERENCES

- ZeroMQ RFC. “7/MDP,” n.d. https://rfc.zeromq.org/spec/7/
- Nlohmann. “GitHub - Nlohmann/Json: JSON for Modern C++.” GitHub, n.d. https://github.com/nlohmann/json
- GeeksforGeeks. “Consistent Hashing System Design.” GeeksforGeeks, October 23, 2024. https://www.geeksforgeeks.org/consistent-hashing/