U.PORTO
FEUP FACULTY OF ENGINEERING
UNIVERSITY OF PORTO

SHOPPING LISTS ON THE CLOUD

LARGE SCALE DISTRIBUTED SYSTEMS

T07 G14

Bruno Rafael Machado (up201907715)

José Francisco Veiga (up202108753)

João Tomás Padrão (up202108766)

MASTER IN INFORMATICS AND COMPUTING ENGINEERING

FEUP 2024/2025

INDEX

01 PROBLEM DESCRIPTION

- **Local-first approach:** The application allows shopping lists to be stored locally on users' devices, guaranteeing data persistence even without an Internet connection.
- **Cloud component:** A cloud-based system for sharing shopping lists between devices and storing backup copies, guaranteeing data availability even if a device is lost or switched off.
- **Unique IDs for lists:** Each shopping list is assigned a unique ID, allowing users to share and collaborate on the list
- **Simultaneous collaboration:** To handle simultaneous updates, the system uses conflict-free replicated data types (CRDTs), ensuring that changes made by several users are merged correctly.
- **Scalability for millions of users:** The cloud architecture is designed to scale, taking inspiration from systems such as Amazon Dynamo, to ensure that access to data is fast and efficient even with millions of users.

02 SYSTEM ARCHITECTURE

- Our system was designed using a **Robust Reliable Queuing** approach, specifically the **Paranoid Pirate Pattern,** which is based on the classic load balancing pattern but improves it with a heartbeat to guarantee reliability and fault tolerance.

Clients:

- Store a local replica of their shopping lists in JSON files.
- Synchronise with the central system when they want.
- Clients repeat operations gracefully if communication fails, inspired by the Lazy Pirate Pattern.

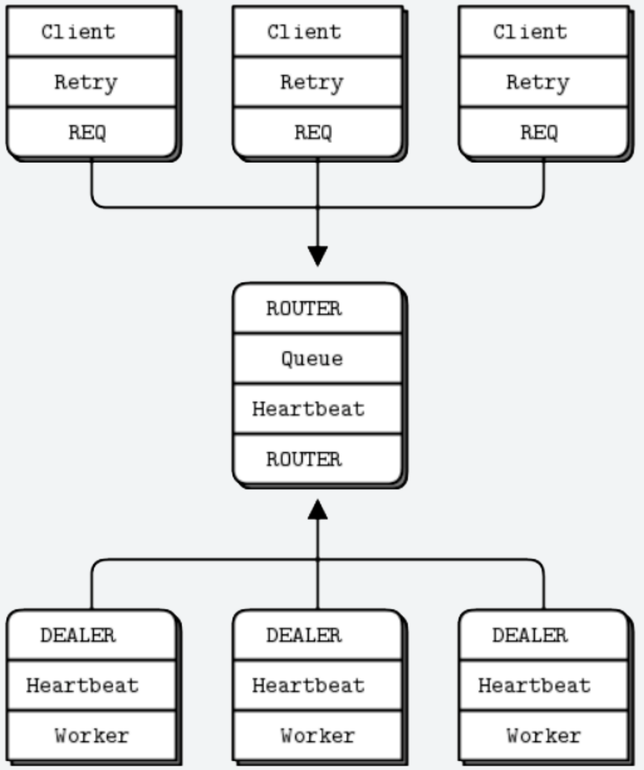

Figure 1 - The Paranoid Pirate Pattern

02 SYSTEM ARCHITECTURE

Broker:

- Acts as a central coordinator between clients and workers. Implemented as a **ZeroMQ router socket**.
- It implements load balancing by dispatching tasks to available workers.
- Uses heartbeats to monitor whether workers are alive and detect unresponsive ones.
- Maintains a queue with workers who are ready to process tasks. When a worker processes a client request, it is removed from the queue until it becomes available again.

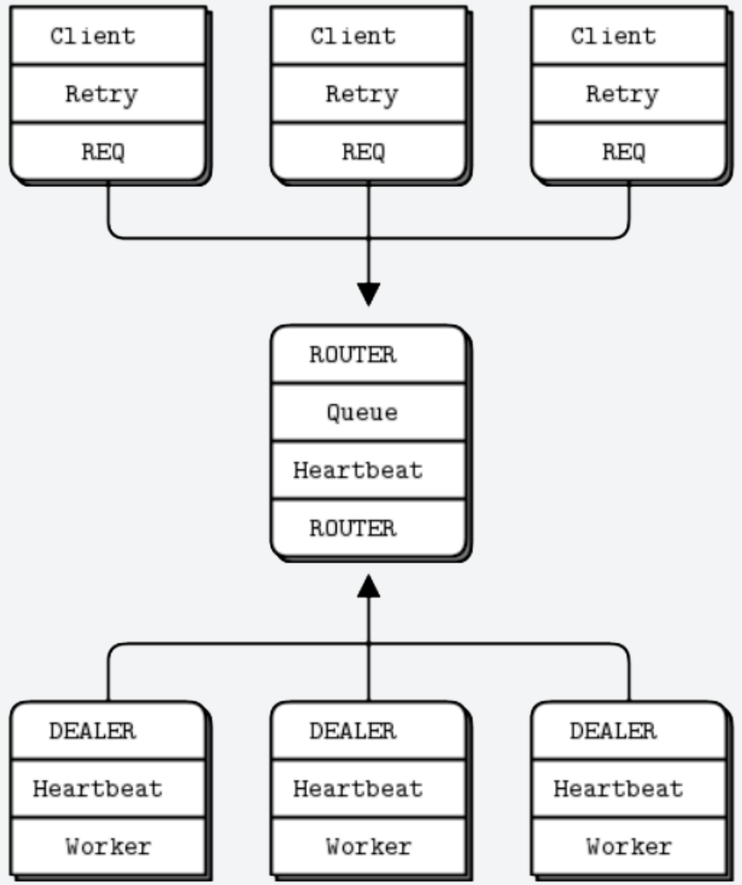

Figure 1 - The Paranoid Pirate Pattern

02 SYSTEM ARCHITECTURE

Broker:

- If there are no workers available, the client's request will remain in the queue until a worker becomes free. To avoid overloading, each request has a maximum lifetime of 30 seconds in the queue, and the queue can hold a maximum of 1000 requests at any one time.

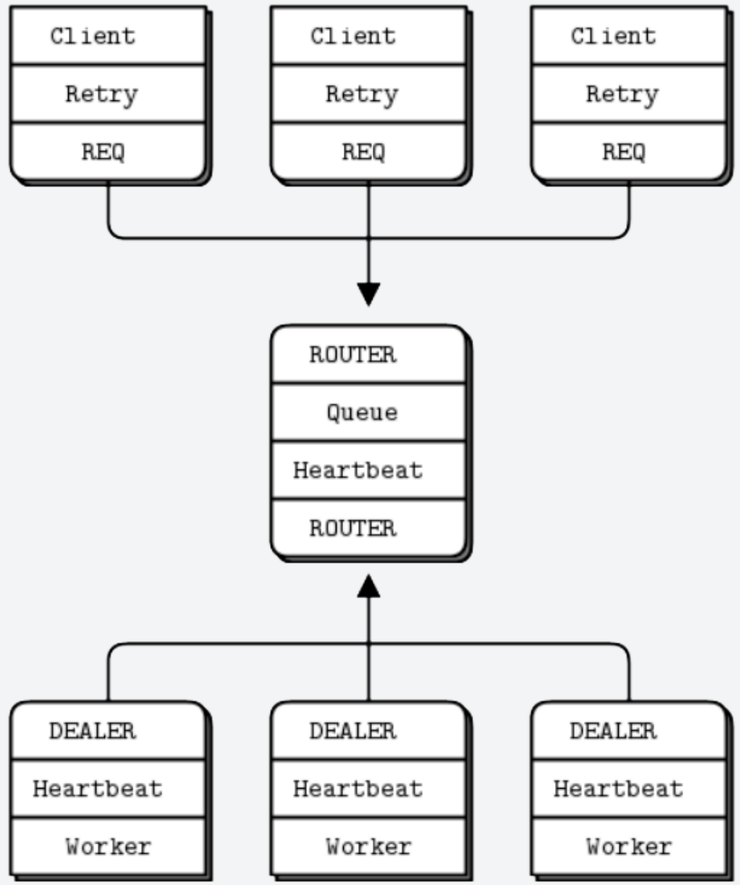

Figure 1 - The Paranoid Pirate Pattern

02 SYSTEM ARCHITECTURE

Workers:

- Implemented as a **ZeroMQ dealer socket.**
- They handle specific tasks, such as updating, retrieving or deleting data from the shopping list.
- They respond to the heartbeats sent by the broker, signalling their availability.
- Use a backoff strategy for reconnections, ensuring efficient recovery without overloading the system.
- Current implementation includes **4 workers.**

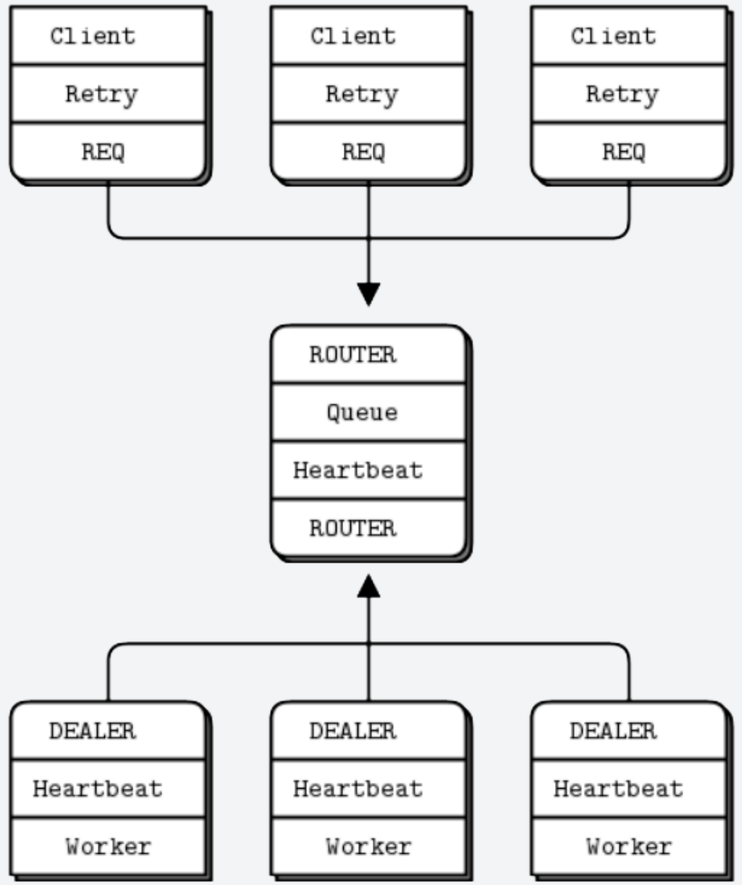

Figure 1 - The Paranoid Pirate Pattern

02 SYSTEM ARCHITECTURE

Database in the cloud (MongoDB)

- Stores shopping lists centrally to enable data sharing and backup between users.

- Chosen over file-based storage for:

1. **Scalability:** handles millions of users with sharding and distributed data storage.
2. **Flexibility:** MongoDB's schema-less nature supports evolving data models.
3. **Performance**: Optimised for simultaneous read/write operations.

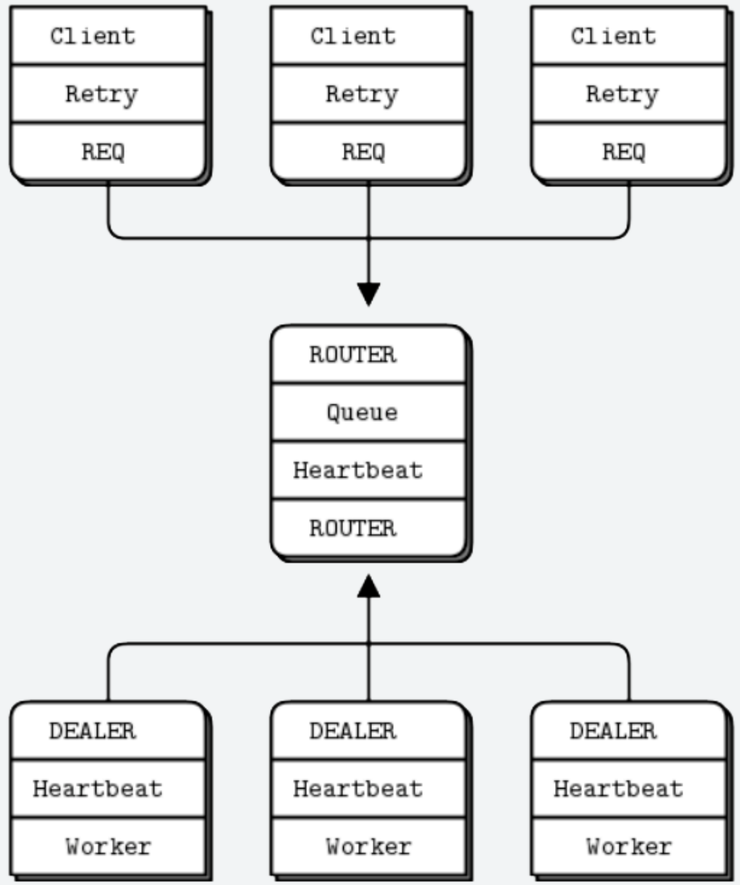

Figure 1 - The Paranoid Pirate Pattern

02 SYSTEM ARCHITECTURE

Advantages of the Paranoid Pirate Pattern

Fault tolerance:

- Detects and handles queue or worker failures through heartbeating.
- Workers automatically reconnect, ensuring that the system remains resilient.

Broker:

- The queue efficiently distributes customer requests to workers, avoiding overloading a single worker.

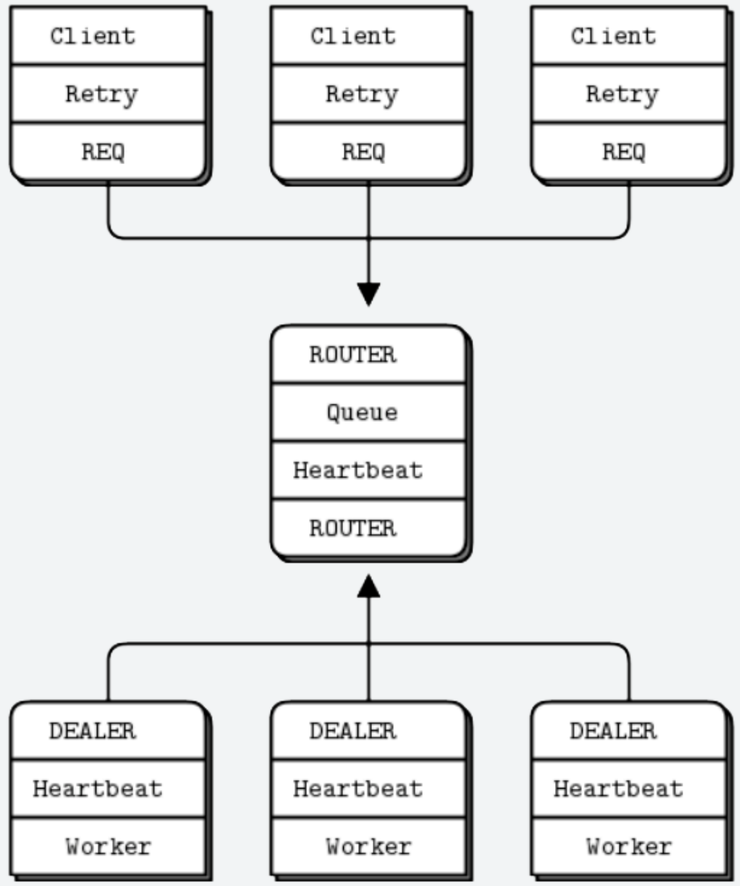

Figure 1 - The Paranoid Pirate Pattern

02 SYSTEM ARCHITECTURE

Advantages of the Paranoid Pirate Pattern

Reliable messages:

- Heartbeats prevent ‘zombie workers’ from remaining in the worker pool.
- Ensures that workers only process requests when they are active and ready.

Scalability:

- Supports the dynamic addition of workers, making the system scalable for high workloads.

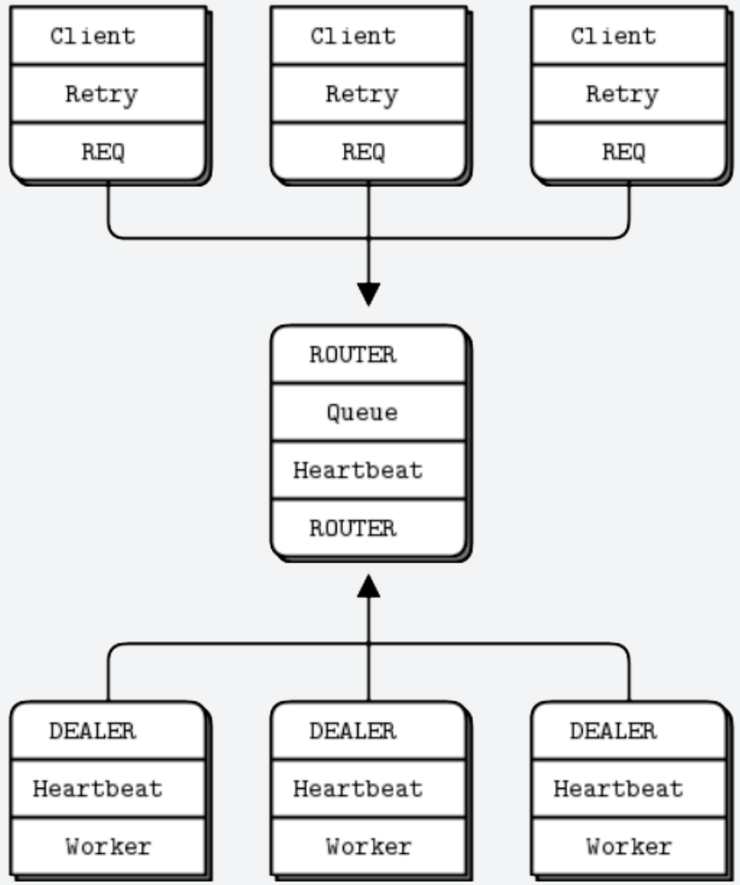

Figure 1 - The Paranoid Pirate Pattern

02 SYSTEM ARCHITECTURE

Disadvantages of the Paranoid Pirate Pattern

Overhead from Heartbeating:

- Continuous exchange of heartbeats between the queue and workers adds network traffic.
- This can become a bottleneck in systems with a large number of workers or frequent heartbeats.

Potential Latency:

- If a worker fails to send a heartbeat within the timeout, there may be a delay before the queue detects the failure and redistributes the task.

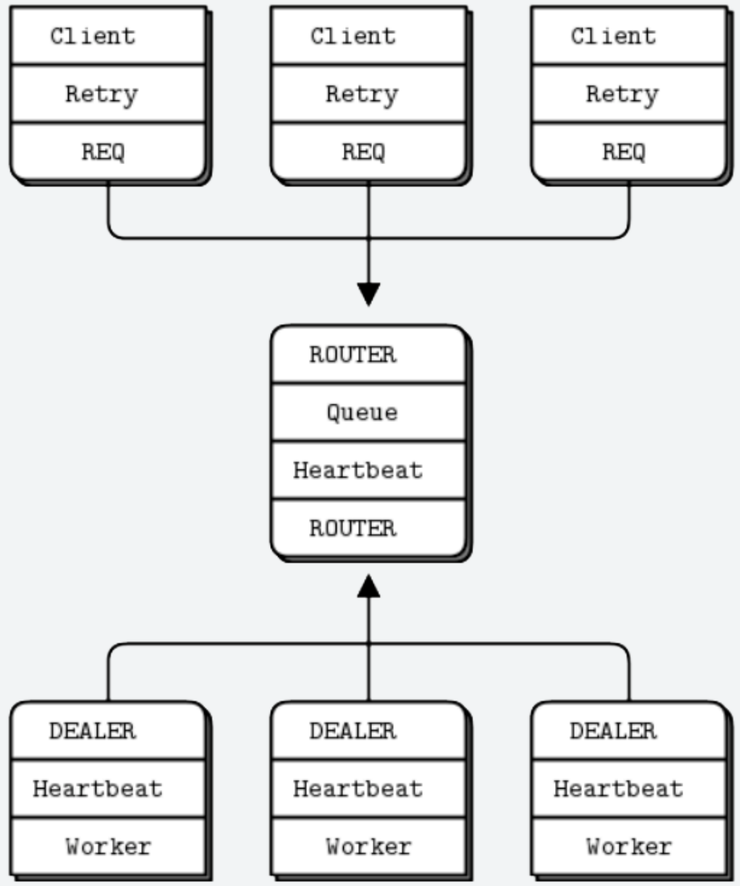

Figure 1 - The Paranoid Pirate Pattern

CRDTS

In this system, we use **conflict-free replicated data types (CRDTs)** to handle simultaneous updates of shopping lists, ensuring that all changes are eventually consistent between conflict-free replicas. CRDTs allow several users to **collaborate** on the **same** shopping list, even when they are working offline, and automatically reconcile the differences when they come back online.

03

CRDTS

CCounter (Causal Counter)

- **Objective:** A counter to record the quantities of items on the shopping list, supporting increments and decrements.

- **Implementation:** Each action in the counter (increment or decrement) is associated with an actor (user) and stored in separate maps for increments and decrements. The state of the counter is updated and merged with other counters to ensure consistency between replicas.

- **Features:**
  - It tracks each user's changes using two maps:
    - **increments**: Stores the number of times an actor has incremented the counter.
    - **decrements**: Stores the number of times an actor has decremented the counter.
  - Supports increment and decrement operations independently.

- **Merge logic:**
  - For increments, it saves the maximum value between the server's counter and the incoming client counter.
  - For decrements, it is similar but guarantees that the true count can't be negative.
  - The real count is simply increments - decrements.

03 CRDTS

ShoppingItem

- **Objective:** Represents an item in the shopping list, along with its associated quantity, which is tracked by a **CCounter.**
- **Operations:** Each shopping item has methods to increment, decrement, and merge its counter based on actions from different actors.
- **Merge Logic:** When two ShoppingItem objects are merged, their respective **CCounter** objects are merged, ensuring that the quantities for the item are reconciled across replicas.

03 CRDTS

ShoppingList

- **Objective:** Represents the complete shopping list, which consists of several ShoppingItem objects stored in a **map<string, ShoppingItem>**.
- **Operations:** The list supports adding items, marking items as purchased (decreasing their quantities) and merging with other shopping lists.
- **Merge Logic:** When merging two shopping lists, the items in both lists are merged. If an item exists in both lists, their ShoppingItem **counters** are merged.

04 CONCLUSION

Conclusion:

- Local-First Design ensures offline access with cloud synchronization for backup and sharing.
- CRDTs (CCounter) maintain consistency across distributed users.
- Scalable Architecture based on the Paranoid Pirate Pattern for fault tolerance and reliability.

Limitations:

- Collaborative Shopping Lists enable real-time updates and conflict-free operations.
- Heartbeating and conflict resolution can introduce some delays in high-load scenarios.

05 DEMO

Demo: https://youtu.be/sn-efuW2-5E