Architecture for cloud-based Shopping Lists application

Work done by:
Dinis Sousa (up202006303)
João Matos (up202006280)
João Pinheiro (up202008133)

Professors:
Daniel Tinoco (Practical classes)
Carlos Baquero-Moreno (Theoretical classes)
Pedro Souto (Theoretical classes)

Design Considerations

Bearing in mind the problem statement, we arrived at the following requirements:

- High availability (for writes): The users should be able to **change any Shopping Lists when needed**. This implies the system should **support concurrency in reading and writing** to the shopping lists.
- Local-first: When writing, the users should always be able to **save their shopping list, even if the server is unavailable**. To ensure that every user should have their database.
- Symmetry: **Every storage host should have the same set of responsibilities** as its peer, to ensure simplicity.
- Decentralization: To ensure there is not just one point of failure, and that the server is always available and running.
- Fault-tolerance: **If** a server or a load balancer **fails**, a **user should still be able to write and read**.
- Scalability: It should be **possible to increase the number of servers** if the number of clients increases, with no harm to the current clients and servers.
- Uniqueness: Every **shopping list should have a unique ID**, that can be shared with other users, and is needed to access a shopping list.

Infrastructure Decisions

Taking Dynamo as our inspiration, we'll implement a **ring-network architecture**, spreading information across multiple storage nodes to achieve consistency, availability, scalability, fault-tolerance, and decentralization, effectively reducing bottlenecks. More details on decisions are below.

- High availability (Resolve conflicts dealt on the reads)
    - **Conflicts will be dealt on the reads**, so no writes are rejected
    - **Sloppy quorum** will improve **availability**, such as a read/write operation is as **slow as the slowest R/W replica**. Will also **improve performance** in **deterioration of consistency**.
- Local first development

    - A User should be **always allowed to write and read** a version of a shopping list **even if the servers are down**, so we will keep record of the events on a **local-storage** before communicating with the cloud.
- Conflict Resolution (Merge conflict logic with last write wins on divergent versions)
    - The first simple implementation would be a **last writer wins** with the help of **local clocks**, later **merge conflict logic** would be implemented to **ease the use** of the application to the **user**.
    - We still plan to implement a **custom counter state-based CRDT** that will make it possible to account for situations where an item may simultaneously receive a quantity decrease, a deletion, and a quantity increase. In this example, we find it more logical to never set the quantity below 0 and then use the value of the increase. We consider **state-based adequate** since the states should not be too large, and are easier to replicate across nodes.
- Work Uniformization through load balancing
    - **More than one load balancer** so our network survives single-points of failure.
    - Load balancers should be **used in parallel** and not only as back-up, to achieve a **highly performance** system.
    - Despite the node selection via consistent hashing of the shopping list ID, load balancers are still important for the **initial routing** of client requests, **handling node failures,** and ensuring **no node becomes overwhelmed**.
- Scalability
    - The **number of nodes** should be able to **increase** to respond to an increase in the number of requests to avoid becoming a bottleneck of the system.
- Redundancy factor (High redundancy factor)
    - **High redundancy factor** to achieve availability at all times for a Shopping-List even though the cost for storage will be higher. This is achieved by having the information of the lists replicated across different nodes.
    - When the app is **scaled to more users, it may reconsider lowering** this redundancy factor.
- Uniqueness (UUID 4)
    - To achieve uniqueness across all servers we will be using **Version 4 of UUID**. A Universally Unique Identifier is a 128-bit label. Their uniqueness does not depend on coordination between the parties generating them, unlike most other numbering schemes. While the **probability that a UUID will be duplicated** is not zero, it is generally considered **close enough to zero to be negligible**.

Architectural Design