Quantum List

FEUP - SDLE

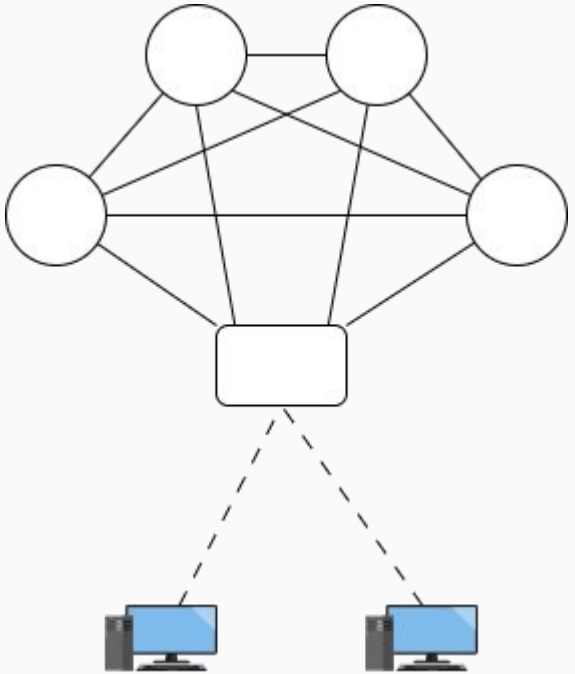
André Ismael Ferraz Ávila - up202006767
Flávio Lobo Vaz - up201509918
Diogo André Pereira Babo - up202004950

Introduction

Quantum List is a local-first shopping list application that stores the lists' data locally and has a cloud component so that not only can data be shared between users but also provides backup storage. The main objective was to build an available, scalable and consistent cloud system that can tolerate errors and handle millions of users. The **application** aims to fulfill as much as possible the seven ideals of local-first: fast, multi-device, offline, collaboration, longevity, privacy, and user control.

Architecture

Our cloud system consists of a cluster of databases and a load balancer that makes the bridge between the client and the cluster while also distributing the load.



Database Cluster

The database cluster consists of multiple nodes, with their data organized in a hash ring, each node is aware of the others and **constantly gossip** with in order to be able update their hash ring, adding new nodes when new nodes are added to the cluster and verifying if a node is alive.

Load Balancer

By the use of a **“bounded consistent hashing algorithm”** on load balancer, we are able to distribute load across nodes in the communication with the ring . Utilizing the replication factor to pass information to the coordinator or communicating with the ring on the addition of a new node.

The Load Balancer also gossips with the nodes to know their state.

We can use also a round robin algorithm.

We can have more than one load balancer working (eg: active/passive to avoid single point of failure), however the detail of using more than one load balancer automatically was not implemented on the client side. It is only possible in the app by entering a new address.

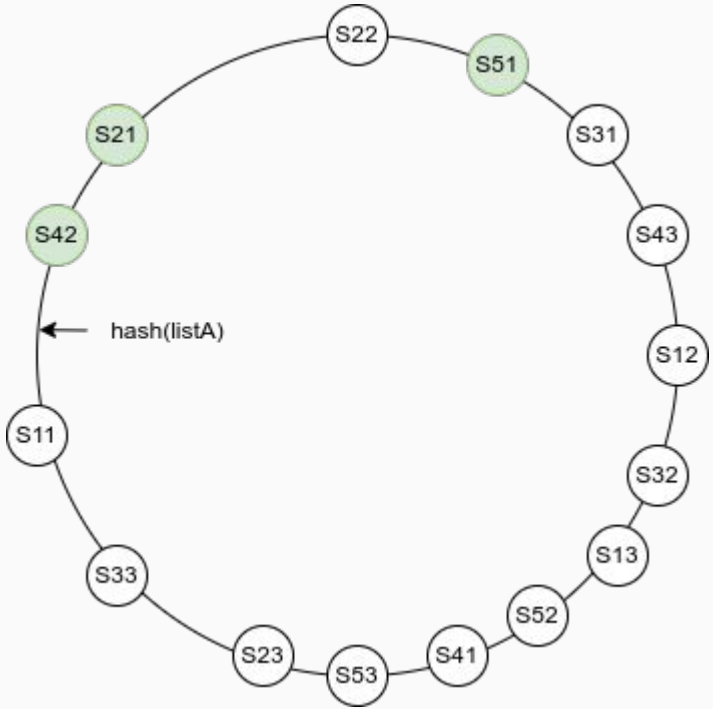
Apart from this the Load Balancer will act as a reverse proxy.

Replication, Data Partitioning and Distribution

Data is distributed between nodes using consistent hashing, once a node receives a request to read or write a list, it becomes a coordinator and it performs a hash to the list id of the given request, using its own own hash ring to determine where should the data be written to or read from. The hash ring has virtual nodes to further distribute the data and avoid bottlenecks. The limit of virtual nodes is 8 as well as the replication factor.

Replication, Data Partitioning and Distribution

The replication factor is decided by the hash ring, and it means that data is replicated to a predetermined number of nodes. The list's id is used to determine the position in the hash ring and the next N (replication factor) healthy physical nodes are selected, clockwise order. A partition is determined by the space between two virtual nodes in the hash ring. On the example, the first number is the physical and the second is the virtual node identifier.



The example above has a replication factor of 3

Replication

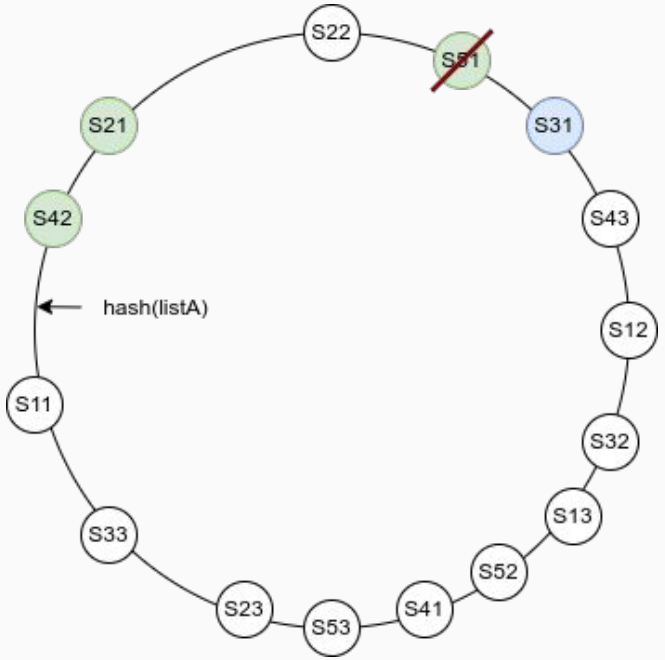
When a node is designated coordinator, he is responsible to ensure the data is written/read, as such he sends to N (replicationFactor/2 +1) nodes a write/read request and await their response, if they don’t respond, the next node on the hash ring is used until it loops. Conflicts are not a problem in our system due to CRDTs. Quorums are performed, and so read-repair is used each read by simply merging the read CRDTs and sending a write back.

A write has two outcomes, written or request failed, one written is enough to be considered successful, as that occasion would mean that every other node is unavailable.

A read has three outcomes, has list and is read, does not have list and request failed, at least one list read is enough for success as the list is then replicated using read-repair, if every outcome does not have the list then the list does not exist.

Replication, Data Partitioning and Distribution

Hinted Handoff is implemented, therefore when the hash ring is updated, data that is written outside of their partition gets sent to every node of their respective partition (being eliminated if at least one node writes it). The hash ring updates every time a node's status changes, if a node does not respond three consecutive times, it is not healthy and therefore not accounted for when selecting nodes.



O exemplo acima tem replication factor 3

Consistency

To guarantee consistency we use state based **CRDTs** (implemented in Rust and Go), allowing concurrent updates by different nodes, with a mechanism to resolve conflicts in a deterministic manner.

Using **PNCounters** limited to values greater than or equal to zero, and a **dot context AWSet**. With this, we create a Shopping List CRDT composed of these . This allows you to register the items you want to buy and those that have already been purchased (the **CRDTs with purchased** items are implemented, but not used on the app), but also remove items, add , increment and decrement quantities, where remove is the operation with least precedence. Only when all users at a given time all remove the list is it deleted.

AWSet, as we will see, using the **dot context** of each list, also allow us to obtain a state for the nodes. With this, we try to have a more suitable **dot context anti-entropy mechanism** for consistent hashing.

Scalability

The cluster consists of multiples nodes (servers with their own database) new nodes can be added at will.

Because we use a load balancer that uses a **“bounded consistent hashing algorithm”** to distribute load according to a chosen threshold (we used 40%), if a node has a total of requests greater than 40% than the node with **fewer requests**, the node chosen by the load balancer will be the one with the **lowest requests**, otherwise is used tnormal consistent hashing to distribute load. Using the above algorithm, Scalability is also reinforced by trying **avoid edge cases**, where a highly requested list from many users, due to having the same hash, would always be redirected to the same node.

We also have the possibility of using the round robin algorithm.

Scalability

The cluster consists of multiples nodes (servers with their own database) new nodes can be added at will.
Because we use a load balancer that uses consistent hashing with a round robin algorithm, scalability is reinforced by trying avoid edge cases, where a highly requested list, due to having the same hash, would always be redirected to the same node.

Fault tolerance

- We have a mechanism of anti entropy, using the hash of the dot context for every list . This allows you to obtain a state for each node. From time to time, each node will check a number of nodes chosen by us **from among those it has replicated (**instead of being for **all possible nodes)** Of these, we choose at random a number of nodes (this quantity of nodes can be defined by us). Then a given node initializes the dot context anti entropy mechanism with those nodes, sending its state (set of list_ids and hash of the list's dot context) . and after some communication between them , they update the appropriate lists between them. This is very important in cases where a node is inactive for some time and becomes out of date.
- Hinted-off

Implementation

The cluster and load balancer were implemented using Golang, the application uses Tauri, which has a frontend in SvelteKit and interacts with a backend in Rust, both application and database nodes use UnQLite, a non relational database that, similar to SQLite, exists on the file system.

The application is local-first, functioning completely with no access to internet, it is fast, and as it is installed by the user, it is available to use offline forever.

Implementation

Regarding our CRDT, due to it not being commonly used we had to create property based testing so that although the consistency was not mathematically proven, we argue that if it works for 1 million different cases, it will serve our purpose.

Conclusion

The work carried out was successful and we accomplished the objective of the project with our architecture. Our implementation of the CRDT also proved to be valuable due to the flexibility and robustness.

Future Work

- Add to application and improve our CRDT that also accounts for purchased items.
- Single point of failure in the load balancer.
- Deal with some nuances regarding our distributed system and a better integration of all the solutions listed here.
- Hinted Handoff could be better, as the solution implemented is less efficient and requires a hash ring update and going through every written list.
- Improve the dot context anti entropy mechanism.