Shopping List Planned Design

Partitioning

One of the key aspects of this architecture is scalability, and, therefore, a method for distributed data storage is required. This architecture employs the Consistent Hashing algorithm to distribute data across various nodes. Additionally, one of the requirements is redundancy, and thus, a variation of this algorithm is used to store the same data on K nodes.

Each node should have a UUID that uniquely identifies it, and using the MD5 hash function, the result represents its position on a circular ring, where the highest hash is connected to the lowest one.

To enhance the likelihood of even distribution of nodes, K virtual nodes are created for each real node.

When a list is added or changed, the load balancer that receives the clients requests uses the MD5 function to calculate the list's position on the ring. Then it will be eventually stored on the next N real nodes (clockwise), skipping the virtual nodes corresponding to the same real node. For instance, as illustrated in Figure 1, if the key falls between nodes A and B, it will be eventually stored in nodes B, C, and D, marked in yellow.

The list is only eventually stored. Actually, the load balancer selects only one real node randomly and sends the list to it. The anti entropy algorithm then propagates the changes made to this node to all the other nodes that make sense given the key's hash.

Recovery from permanent failures

The failure of nodes does not restrict the user from accessing and editing their shopping list since the information is replicated across multiple nodes. However, when a node fails permanently, the preference list of each key it used to store must be modified:

- If said node was the coordinator of any shopping list, the new coordinator becomes its first neighbor (clockwise);
- The (new) coordinator node of each key range that was stored on the failed node must replicate its keys to its (N-1)th new successor node (clockwise).

As shown in Figure 2, node C had a permanent failure and it was storing the shopping lists whose keys fell in the ranges (F, A], (A, B], and (B, C], which means that it was the coordinator for the keys that fell between nodes B and C. After removing it, we can verify the following events:

- the node D becomes the coordinator for the keys on the range (B, C];

- the node A sends to node D the shopping lists whose keys are in the range (F, A];

- the node B sends to E the lists with keys that fall in the range (A, B];

- the node D sends to F the lists whose keys are in the range (B, C].

Fault tolerance

The load balancer stores a list of active nodes (i.e., nodes that haven't failed permanently) and a list of nodes that are considered to be in the midst of a temporary failure. For a given key and shopping list, a random node from the list of nodes corresponding to the key, as explained in the partitioning section, that isn't present in the temporary failure nodes list is selected. After this, a temporary failure timeout and a permanent failure timeout are activated. The temporary failure timeout is activated implicitly by using TCP Keep-Alive, while the permanent failure timeout is activated explicitly.

Once the temporary failure timeout is exceeded and the TCP connection is closed, the node is added to the temporary failure nodes list and the load balancer tries to store a shopping list in another node from the list of nodes corresponding to the key that isn't present in the temporary failure nodes list, and so on and so forth. Note that if, for example, the node that the load balancer is attempting to connect closes the connection, the node is also assumed to be failing temporarily.

The permanent failure timeout is deactivated whenever the node sends a confirmation that it stored the list or when it reestablishes a connection with the load balancer. If the node doesn't reestablish a connection and the permanent failure timeout is triggered, then the node is removed from the token ring as explained in the recovery from permanent failure section. The initial TCP connection will have been necessarily closed by then, as the permanent failure timeout is longer than the Keep-Alive timeout.

Whenever a node is in the temporary node failure list, the load balancer and the anti-entropy algorithm will both skip it when selecting the N real nodes to store the list and the N-1 neighbors, respectively. If the node reestablishes the connection before the permanent failure timeout, then it is sent all the data that it has missed from its neighbors.

CRDTs

For dealing with the conflicts that arise from multiple users editing the same shopping list, we decided to create a shopping list level CRDT: ORMap<id, CCounter>. This CRDT will map each item, with a certain id, in the shopping list to its target quantity, which can increase or decrease and, therefore, makes sense to be represented as Counter. For the Counter, since the increase and decrease operations commute, we don't need to be worried about the final state reflecting the effect of the registered operations. For the map, we will follow a write-wins semantics, meaning that if one user updates one of the items in the list while another deletes the same item, we will preserve the updated item in the list to make sure data isn't lost.

Client Synchronization

In order to keep the shopping lists of the clients connected to the system always updated, we decided to add a pub/sub component. Each client will subscribe to the updates of their shopping lists and, each time a write is processed in the servers, the updates made to that shopping list will be sent to the local copies of the connected clients subscribed to that list. In case a client isn't connected at the time an update is made, the synchronization will occur once the client connects again to the system, before any request is processed.

Replicas Synchronization

Our system will implement an anti-entropy protocol to keep replicas synchronized. Periodically, for each node X, a random node Y with which X shares a key range is selected. A map from keys to the hash of the corresponding shopping list is sent by X to Y containing all keys in the shared range. Y replies with keys in X but not in Y and a map from keys in Y but not in X and keys for the different hashes received to the corresponding shopping lists. After receiving this map, node X merges the shopping lists that had a different hash, stores them along with the shopping lists for the new keys from Y and sends the merged shopping lists back along with any shopping list whose key was not present in Y.

Annexes

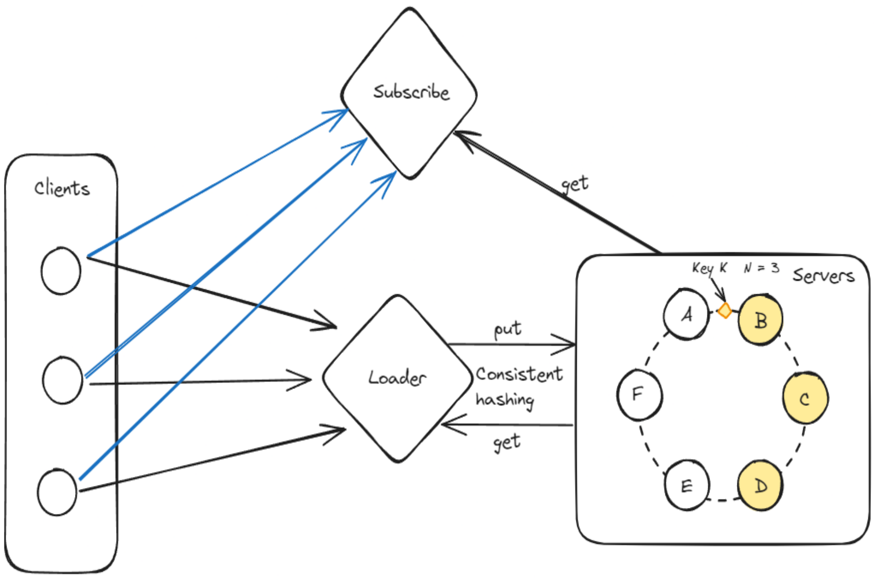

Figura 1: System Architecture

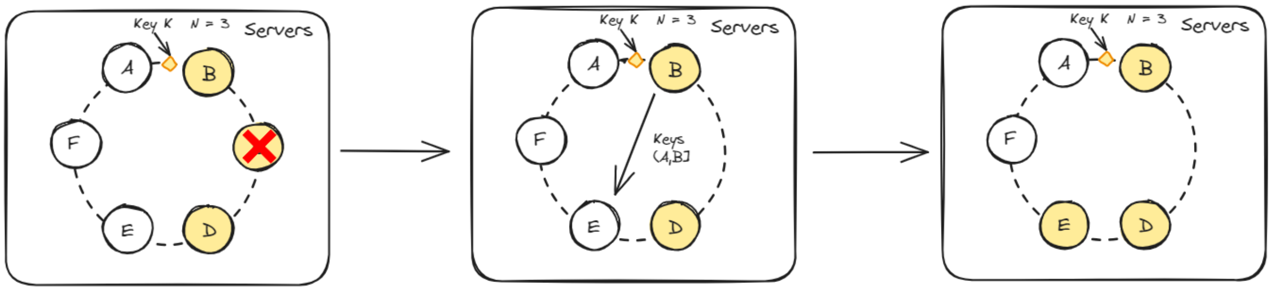

Figura 2: Failure Recovery