Large Scale Distributed Shopping List Architecture

Large Scale Distributed Systems

Ricardo Inácio[1] Diogo Lemos[2] Tomás Maciel[3]

FEUP
{up202302742[1], up202003484[2] up202006845[3]}@up.pt

1 System Architecture

For our system's architecture, it was decided that clients and servers must communicate through a proxy, that should act as an intermediate, executing the following procedure when reached: take a list's unique key, and use a **hashing** function to generate the hashed key. It will then use a **lookup** function that is responsible for finding the server to which the information will be sent or retrieved (the server's position is also determined by its key's hash). This is done on a "ring of servers" in which each one is responsible for storing and replicating a range of keys. With this implementation, we'll be able to deal easily with server failure, because each data item is replicated at N hosts, where N is a parameter that we set "per-instance". The coordinator will store locally the key of its range, and then replicate it to the N-1 next nodes in a clockwise direction. Divergent versions of the same shopping list (with the same ID), must be merged, and stored at both client and server levels: when a client first connects with the server (through the proxy), it will receive an already merged list; when a divergent version (from a local list) arrives at a client, it must merge them based on vector clocks; when the server receives divergent lists, it must perform the merge internally, and then replicate to other nodes the new version.

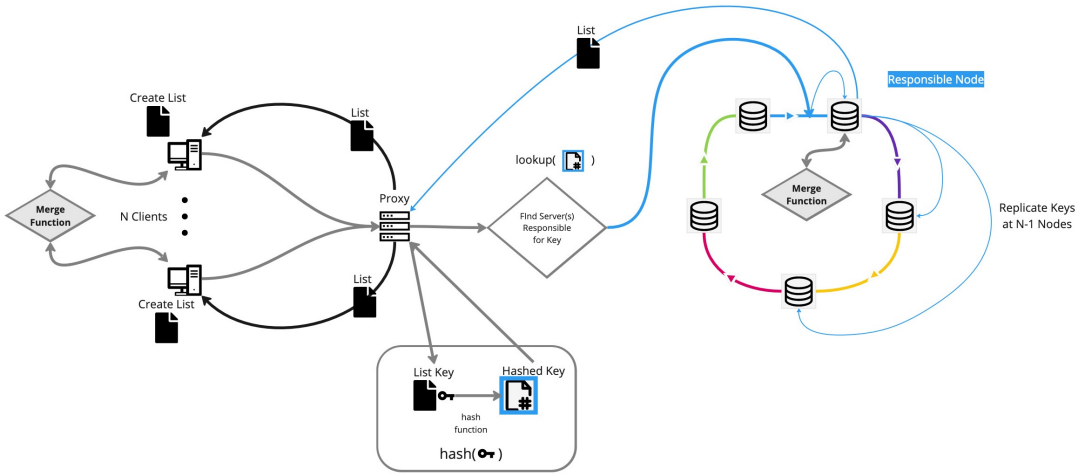

Figure 1: System's Architecture Diagram

1.1 Consistent Hash Ring

We chose a consistent hash ring to evenly distribute the data across the nodes in the system. The greatest upside of using it, is in case one node goes offline, the data assigned to that node is redistributed to the next one without harming the others, and if a new one joins, it gets assigned some data from the next node. It guarantees load balancing, scalability, and fault tolerance while being deterministic, which means that since the same input will always produce the same output, there is no need for all nodes to recompute or relearn values in case of failures. Also, this scheme is independent of the number of participant nodes or servers. As they join, their position (in the ring) must plainly be chosen based on the hashing of a distinguishing key, and as they leave, all the data keys associated with them must merely be relocated to the next found node, in a clockwise direction.

1.1.1 Hashing Algorithms

Hashing algorithms are one way to convert data into fixed-length hash values, while using **uniform distribution**, which guarantees that keys are evenly distributed among the available nodes in the key space, because since position is specified by the hashed value, it's impossible that one area gets assigned more elements than others. Using this concept, the **lookup** function will be extremely efficient, since there's simply the need to hash the desired value and search for it in the preference list, containing its responsible nodes. One of these algorithms is MD5. It generates a 128-bit hash value from input data, which allows it to be fast and efficient for data integrity identification and checksums.

1.2 Conflict Resolution

To account for the concurrent nature of the system, whilst maintaining high availability, the **eventual consistency** model was chosen, more specifically, the **Last-Writer-Wins** implementation. Using this model, there is a guarantee that sometime in the future all inconsistencies will be resolved, on the basis of **merge** operations. Participants in the communication process must send the latest versions of their local data (as they alter it, or asynchronously), so that eventually the system may **converge**, which is known as **anti-entropy**[2].

1.2.1 Last-Writer-Wins

This method allows the group to use **sets** to store the shopping lists, as well as their items, as they are added or removed from respective lists. Alongside the items, a **timestamp** is attached, so that conflicts can be sorted out, during **merge** operations. It takes the form of an **element-set**, which namely contains the *add* and *remove* sets, related to their operation-respective items. By comparing both sets, and if needed, the elements' timestamps, an item can be considered part or not of the merged set. In case of ambiguity, more relevance can be stated to either *add* or *remove* decision bias.

1.2.2 Merge Operation

When divergent versions of the same data are identified by a server or client, a **merge** must resolve said conflict. This is achieved by comparing add (A) and remove (R) sets, as well as the item's timestamp. If an item is in A and not in R, then it's part of the set, if it's in R and not A, it's discarded. But if it's in both A and R, it is only part of the set if the item's timestamp in A is higher than R's (meaning it was added, removed, then added again). If the same item is in different A sets with different timestamps, then only the maximum value prevails. The timestamps used need to be coherent and comparable by different nodes in the network, making the use of real/local clocks impossible, so to account for this factor, the use of **vector clocks** was decided, to ensure partial-ordering, which is sufficient for the system requirements.

1.2.3 Vector Clocks

This data structure not only ensures partial ordering, but also detects causality violations [1] and verifies consistency. They operate in a decentralized manner, where each node in the system keeps track of its own logical clock, which allows them to establish "version" control. Vector clocks elements are in the format *(N:T)*, in which N is the node identifier and T is an integer that, in this case, will represent the version (in which a higher value represents a more recent version of the shopping list). With this, there's an indicator of the causal time a change was made, which allows the server to save a correct merged version. Just an example, if we have a *VC[N1:2, N2:1]* at **N1** and a *VC[N1:0, N2:3]* at *N2*, when eventually a merge happens (let's say, at *N1*), a new version (*VC[N1:3, N2:3]*) will be stored, and then replicated, so no one is harmed, and the conflict is easily solved.

References

[1] Roberto Baldoni and Matthias Klusch. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(02), 2002.

[2] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12, 1987.