Shopping List on the Cloud Architecture

Gonçalo Domingues
up202007914@up.pt
Faculdade de Engenharia do Porto
Porto, Portugal

José Araújo
up202007921@up.pt
Faculdade de Engenharia do Porto
Porto, Portugal

Marcos Aires
up202006888@up.pt
Faculdade de Engenharia do Porto
Porto, Portugal

ABSTRACT

This project outlines the development of an efficient architecture for a distributed shopping list system. The proposed system aims to enable real-time collaboration and synchronization of shopping lists across various devices, emphasizing scalability and robust data consistency.

KEYWORDS

Distributed Systems, Shopping List Management, Real-time Collaboration, Data Consistency

1 INTRODUCTION

This project will explore the creation of a local-first shopping list application. The application has code that runs in the user's device and can persist data locally. It also has a cloud component to share data among users and provide backup storage. In response to the need for a solid and scalable shopping list management system, for this project, we propose a cloud architecture based on the proven infrastructure of Amazon's Dynamo. By resorting to Dynamo's reliable and scalable features, this architecture aims to enable easy collaboration and synchronization of shopping lists across many devices. The incorporation of Dynamo's practices ensures efficient data management, emphasizing scalability and data consistency. This research seeks to explore the practical integration of Dynamo in the context of developing a distributed shopping list system.

2 LOCAL

The distributed system we're going to develop is a local-first system, i.e., all the changes a client wants to make to a determined shopping list in the cloud need to be written locally first. When a client wants to make a change to a shopping list, he first needs to create a local copy of the one he wants to change and write the changes to it. Only after that can the changes be sent to the cloud. This is made so that the client can still make changes if he is offline, as this changes will persist locally (even though the changes are always first written locally, whether the client is online or offline). When the client is online again, the changes can then be sent to the cloud. As can be seen in the architecture in the annex, each client has a local copy of the shopping lists that he wants to change. Only after this will a server that stores that changed list in the cloud receive a request to store the modifications. This will happen by using a REQ/REP mechanism with a router with a load balancing broker. When a write is to be performed, the router will invoke the put() method and the load balancer will apply the hash function to the object and decide which server will be the coordinator for that write (which node will receive it and send to the other nodes who have replicas of that object). As dynamo uses a configurable quorum-like protocol with parameters R and W, at least W-1 nodes need to receive the replica before the write is considered successful. A similar thing happens when a client wants to read a value from the cloud: the router will invoke the get() method and the coordinator node for the object requested will request the latest versions of an object from multiple replicas. It waits for at least R-1 responses before returning a result to the client. If conflicting versions are returned to the client, he will handle the conflict resolution and subsequent updates.

Additionally, since users can concurrently change the lists and we aim for high availability, we will first use Last-Writer-Wins with local clocks, which means that only the last modifications made to a list will be considered, and the other ones will be discarded. Only later we will evolve to use Conflict-free Replicated Data Types, which would allow both changes to be considered (the shopping list would have, in the case aforementioned described, one banana).

3 CLOUD

3.1 Data Partitioning

Dynamo performs data partitioning (sharding) of the data across all the nodes (servers), present on the cluster. Also, to ensure the availability of the system, Dynamo replicates a configurable number of replicas of the data partitions to other nodes, in order to have some redundancy.

3.1.1 Consistent Hashing. In order to know where each data partition stays in the cluster, Dynamo uses a consistent hashing algorithm. Looking at the architecture in the annex we can better understand how consistent hashing works. First of all, a hash space is mapped in the hash ring (for effects of demonstration, we used the hash space in the architecture draft goes from 0-99. Values are merely illustrative). Then, all the servers are hashed using a hash function and are mapped on the ring, and that same hash function is used to calculate the hash for the keys of the object and attribute them to the servers across the ring (an object's key belongs to the server it is closer to). As can be seen in the image, k1 belongs to server B, k2 to server C, and so forth. Consistent hashing is independent of the number of servers in the hash ring. So, only a small number of objects need to be redistributed when there is a change in the number of servers. For example, if we remove server B, we just need to redistribute the key K1 to the next server in a clockwise direction. In an analogous way, if we add a new server between the servers D and A then we just need to redistribute the key K1 to the newly added server. Hence, consistent hashing allows elasticity in the cluster with minimal object redistribution.

3.1.2 Replication. Replication is an extremely important mechanism in Amazon's Dynamo architecture. Observing the architecture at the annex, we can take a better look at how replication works. Imagine we have a replication factor (N) of 3. This means that a key of an object will be replicated to N-1 nodes (servers) clockwise from the node it belongs to, additionally to already being in the node it is assigned to. So, as we can see in the image, k1, which belongs to server B, will be replicated in servers C and D, while k2, which belongs to server C, will be replicated in servers D and A. This replication ensures redundancy and allows for a high availability and fault tolerance in the system, as we're going to see briefly.

3.2 Reading and Writing

3.2.1 Vector Clocks. Data systems with eventual consistency, like Dynamo, may consider a write operation as complete before it propagates to all replicas, potentially leading to conflicting data across nodes. This approach can result in older data being returned during subsequent reads, especially in the presence of network issues or node failures. To manage these conflicts, our system will be using Vector Clocks, which help track the causality between different versions of data. Each node maintains its own vector clock, an array of counters associated with different nodes in the system, to detect and resolve conflicts within the distributed environment. Vector Clocks serve as a mechanism in distributed systems to monitor the order of events or updates across nodes. They become particularly useful when concurrent updates occur on multiple replicas of the same data, preventing inconsistencies and aiding conflict resolution. During communication or synchronization between nodes, the exchange of vector clocks helps identify the causal relationship between different updates. Comparing these vector clocks allows systems to determine the relative ordering of events, which is essential for maintaining data consistency across multiple nodes in the distributed system.

To illustrate the application of vector clocks, consider Figure 3's example from the annex. Initially, a client writes an object, resulting in Object 1 and its clock [(A, 1)] when it reaches the server node A. Assuming another update by the same client but handled by a different server (B), the system now has Object 2 and its clock [(A, 1), (B, 1)]. In a different scenario, if a different client reads and attempts to update Object 1 and a separate server node (C) performs the write, the system records Object 3 with the clock [(A, 1), (C, 1)]. Nodes aware of Object 2 and receiving Object 3 will detect no causal relation between them, necessitating the preservation of both versions for reconciliation/merging upon a read. Let us suppose that a client reads both Objects 2 and 3. If the client merges and node A coordinates the write, for example, the updated data Object 4 adopts the clock [(A, 2), (B, 1), (C, 1)].

3.3 Membership

Our system is meant to be a distributed and decentralized system, where nodes must rely on peer-to-peer communication to ensure information integrity and consistency.

3.3.1 Gossip Protocol. Like Dynamo, we will be utilizing the Gossip Protocol to disseminate information among nodes. This protocol involves nodes randomly selecting another known node at regular intervals (every second, for example) and exchanging information. In case of conflicting information, the nodes will sort out their differences. This process continues until all nodes in the cluster eventually agree on the same information.

3.3.2 Update Membership (Adding and Removing Nodes). In our project, every node (server) is supposed to know about the existence of all the nodes in the cluster, keeping a catalog of them in a list-like data structure containing all nodes in the cluster. When adding or removing a node, membership changes, and this change in membership is then communicated to the entire cluster via the Gossip Protocol. To provide an example based on the image in the annex that represents our architecture, imagine a given node E joining the cluster. Node A, the initial point of contact, updates its own membership adding node E to the list and shares this update with node B. Node B, after receiving the information about node E, also updates its local membership list. The process continues as nodes A, B, C, and D, in consecutive iterations, communicate their membership changes to other nodes through random selections, ensuring that all nodes are informed about the new cluster. When a new node starts for the first time, its position within the consistent hash space is selected by the hash function and it uses the Gossip Protocol to distribute this information. Over time, all nodes within the cluster become aware of the added node (node E) and the specific data ranges it is responsible for handling.

3.4 Handling failure

Dynamo was designed, from the beginning, with the expectation that things would go wrong. So, besides replication, there are other techniques it uses to handle failures.

3.4.1 Sloppy Quorum and Hinted Handoff. Looking then again at our example, let's imagine the server receives a request to write a value with the key k1. As it can be observed, the key k1 is stored in the server B. So, as the replication factor in our example is N=3, the key will be replicated to the nodes C and D. Additionally, we will consider a W=3, which means that at least the coordinator and another node must acknowledge the write before it is considered successful and the node (server) B returns to the caller. However, let's assume the data center hosting server C suffers an outage. Instead of aborting the write, B will walk the ring in a clockwise direction and replicate the key to the nodes D and A. When one of these nodes acknowledges the write, it is considered successful and node B can finally return to the caller. This is called a "sloppy quorum". As node A was not supposed to receive the replica, it stores the replica on a separate database and receives a hint which specifies which node was supposed to have the replica. Knowing it was supposed to be node C, A constantly checks it to know if it is back up again. As soon as C gets back up, A delivers the replica to it and deletes the object from its local storage. This is known as a "hinted handoff".

3.4.2 Merkle Trees. Merkle Trees are the mechanism that Dynamo uses in order to recognize any inconsistencies between nodes. Individual nodes maintain separate Merkle Trees for the key ranges that they store and they can be used to quickly work out if data is consistent and if not, identify which pieces of information need to be synchronised. This allows the Dynamo architecture to be resistant to permanent node failures, because, if a node is down for an undetermined time, the data it contains can be fetched from any other node, as they are all synchronized. A Merkle Tree is a tree in which leaf nodes represent the hashes of individual data items and every subsequent parent of those nodes is constructed by hashing together the concatenated hashes of its children. Let's say that the Merkle Tree shown in the annex (which is merely illustrative, hence all hash values in it are fictional and invented off the top of the head) is maintained by node A and we're comparing it with node B. In order for the comparison to take place, node A might request node B's root hash. When it receives the latter, it compares its root hash with it and, if they match, we can be sure that both nodes contain the same data and are, of course, synchronized. But, for the sake of explaining the importance of this structure, let's assume that the hashes don't match and node B's root hash is 22l0b2f6. Then, the hashes for the level below the root are requested from node B which responds with 7495S7YT and 94i311DG respectively. The first hash, 7495S7YT matches the left side of node A's tree, but the second hash is different. This means that the data on the left side of the tree is the same on both nodes. So, node A would then ask B for the hashes for nodes under 94i311DG. Node B would then respond with C512D5346 and 7b8DBDFA. The first hash matches the hash(C) in node A's tree but the second hash does not match the hash (D). This indicates that the value for the key that has a hash(D) must be updated. So, in conclusion, this mechanism avoids the need for nodes to send entire datasets to other nodes in order to compare the values of the replicas of its objects, allowing a reasonably small overhead in the implementation.

4 CONCLUSION

Drawing inspiration from Amazon's Dynamo, integrating its key features and mechanisms allows us to build a strong architectural solution for our distributed shopping list system. Adopting concepts like consistent hashing and the Gossip Protocol, we ensure scalability, fault tolerance, consistency, failure detection, and recovery. Implementing these Dynamo techniques ensures a robust foundation for our distributed system designed to meet the specific needs of our shopping list application.

5 BIBLIOGRAPHY

[n. d.] Amazon's dynamo paper. (). https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf.

6 ANNEX

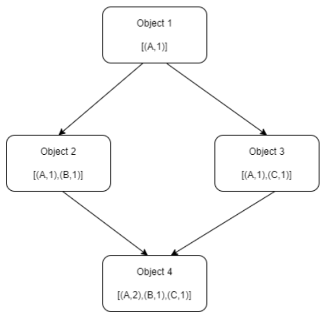
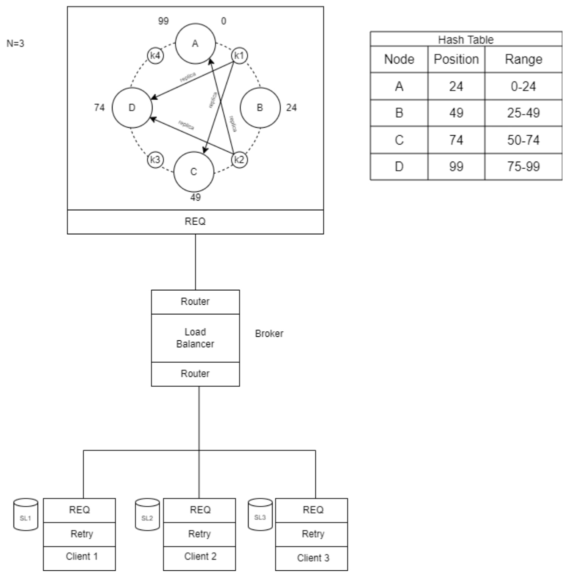


Figure 3: Vector Clocks



Hash Table

Node	Position	Range
A	24	0-24
B	49	25-49
C	74	50-74
D	99	75-99

Figure 1: Shopping list system architecture (REQ is to be applied to each node. The way it is in the image is just for convenience purposes)

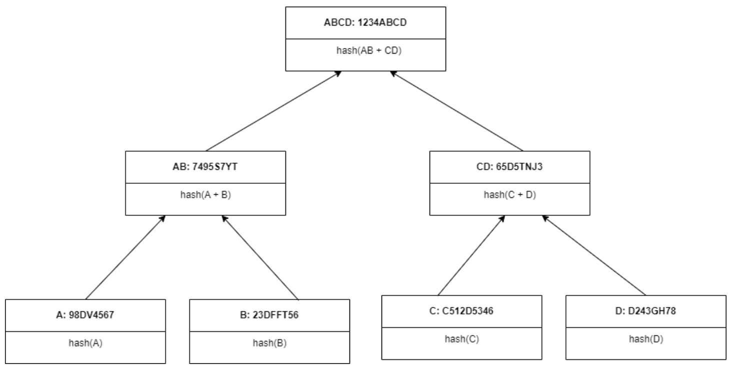


Figure 2: Illustrative Merkle tree