Dynamo

Final report

Yishen Sun
yishen.sun@wisc.edu
University of Wisconsin-Madison

Yun-Chen Tu
ytu28@wisc.edu
University of Wisconsin-Madison

May 21, 2023

1 Design

1.1 Dynamo introduction

Dynamo is a highly available key-value storage designed by Amazon [1]. The main focus is to give clients an "always-on" experience. Under the CAP theorem, the design focuses on availability and partition at the cost of consistency. Compared to Raft, Dynamo doesn't have a leader server to keep request order and maintain strong consistency, while every peer can serve as a "coordinator" if they take responsibility for those keys. Five major techniques are introduced to solve potential problems. First, the consistent hashing technique is used to distribute keys among different servers. The key is hashed and assigned to specific groups of servers to handle it. Second, vector clocks with reconciliation during reads are introduced to improve performance. It's possible that the system experiences temporary failures or other problems, resulting in conflicting records. To improve performance, the system allows users to write data and delay resolving the conflict records. Third, sloppily quorum and hinted handoff are introduced to handle temporary failures. Given that a key is assigned to three servers (A, B, C), server D or another server may take responsibility for the data if one of them is temporarily down to maintain high availability. Fourth, a gossip-based membership protocol is introduced to manage membership and detect node failures. Lastly, anti-entropy using Merkle trees is introduced to solve permanent failures. The hash trees are maintained on each server, and they can verify the correctness of records by passing the hashing information instead of all the data they have, which improves the cost of comparing that data.

We implemented the first four techniques in this project. The system is composed of client and server parts. The client library enables clients to read or write data into the system, while the server library enables users to run peer servers to handle requests from clients, administrators, and other servers. The administrator library allows administrators to notify the existing network that a new physical node will join the network or that a failed node has restarted and will resume its responsibility for partitioning data. Each library communicates with each others using gRPC. The whole project is implemented in C++, and we only store the key-value data in a standard unordered map library for the state machine.

1.2 gRPC design

gRPC includes three categories. One category is for communication between the client and server libraries. The client triggers either a put or get request to the servers, and the servers take further steps and return the results. The second category is designed for communication between servers, including peer put, peer get, and gossip functions. As servers receive requests from clients, the server and coordinator collect responses from the quorum and return the results. Each server also periodically sends a gossip message to other servers to recognize whether any server is unavailable. The third category is for administrators to notify the existing network that a new node has been added and to trigger each node to maintain the latest consistent hashing ring and move data to assigned servers.

1.3 Client library

The client library includes two major APIs for put and get operations. The client library maintains an in-memory cache for data, including key-value pairs and their versions. When the put operation is triggered, the client-side takes responsibility for attaching the version of the data to maintain consistency. If there is no data information in the cache, the client object starts by reading the data from the system to receive the most recent version. The other operation is put operations, and the client library takes responsibility for reconciling the data if there are conflicting records existing in the system. We follow the "read-repair" design. As the client reads the

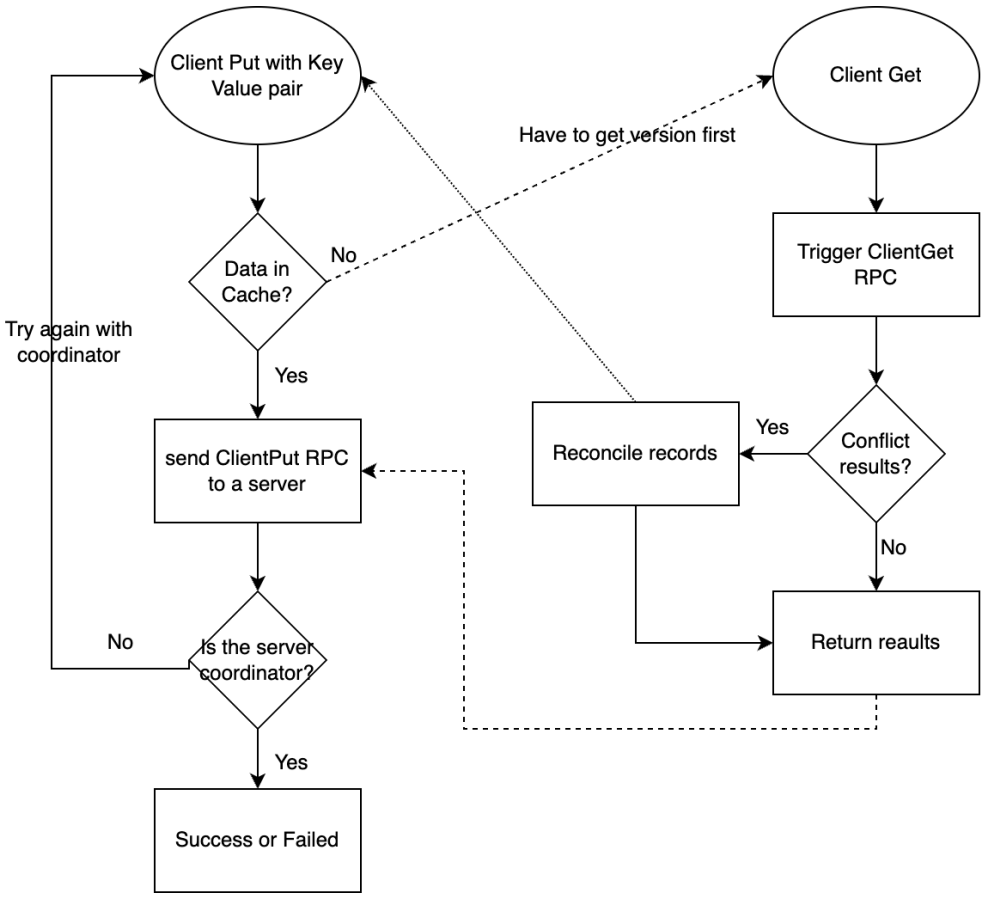

Figure 1: Client-side function workflow diagram

conflicting records, the library asks the user to select one data first before storing the new data.

1.4 Server library

The server library used in our system is divided into three parts: communication between the client side, the peer side, and the administrator side. This library also includes an important implementation of consistent hashing ring, which enables peer servers to determine the range of keys they should handle.

For the get operation, when the server receives a client request, it takes responsibility for collecting information from all quorum members that are handling the requested key. On the other hand, the server performs differently for put operations. If the server is the coordinator, it updates the version of the data object and informs all the remaining quorum members about the new data. However, if the server is not part of the quorum, it instructs the client about the correct peer to communicate with and asks the client to retry the request (fig.2).

In the current design, we replicate the data onto three machines, ensuring redundancy and fault tolerance. Both read and write operations require at least two acknowledgments from the quorum members before returning a response to the clients. This ensures that data consistency is maintained and that the operations are performed reliably.

Each node in the network continually initiates gossip RPCs. Initially, a node updates its own heartbeat timestamp within the members' heartbeat list. Following this, it constructs a GossipRequest message encapsulating the node's membership list, where each entry constitutes a key-value pair comprising the node ID and its most recent heartbeat timestamp. This request is subsequently dispatched to other nodes in the network. A node transmits a gossip message to another node only if their last interaction falls within a specific time interval. If a node encounters a failure while initiating the gossip RPC, it generates an error message.

Upon receipt of the RPC, other nodes in the network update their respective members' heartbeat lists. Specifically, they replace their local timestamp with the newer timestamp, whether it's the locally stored one or the one received in the RPC. This process ensures that every

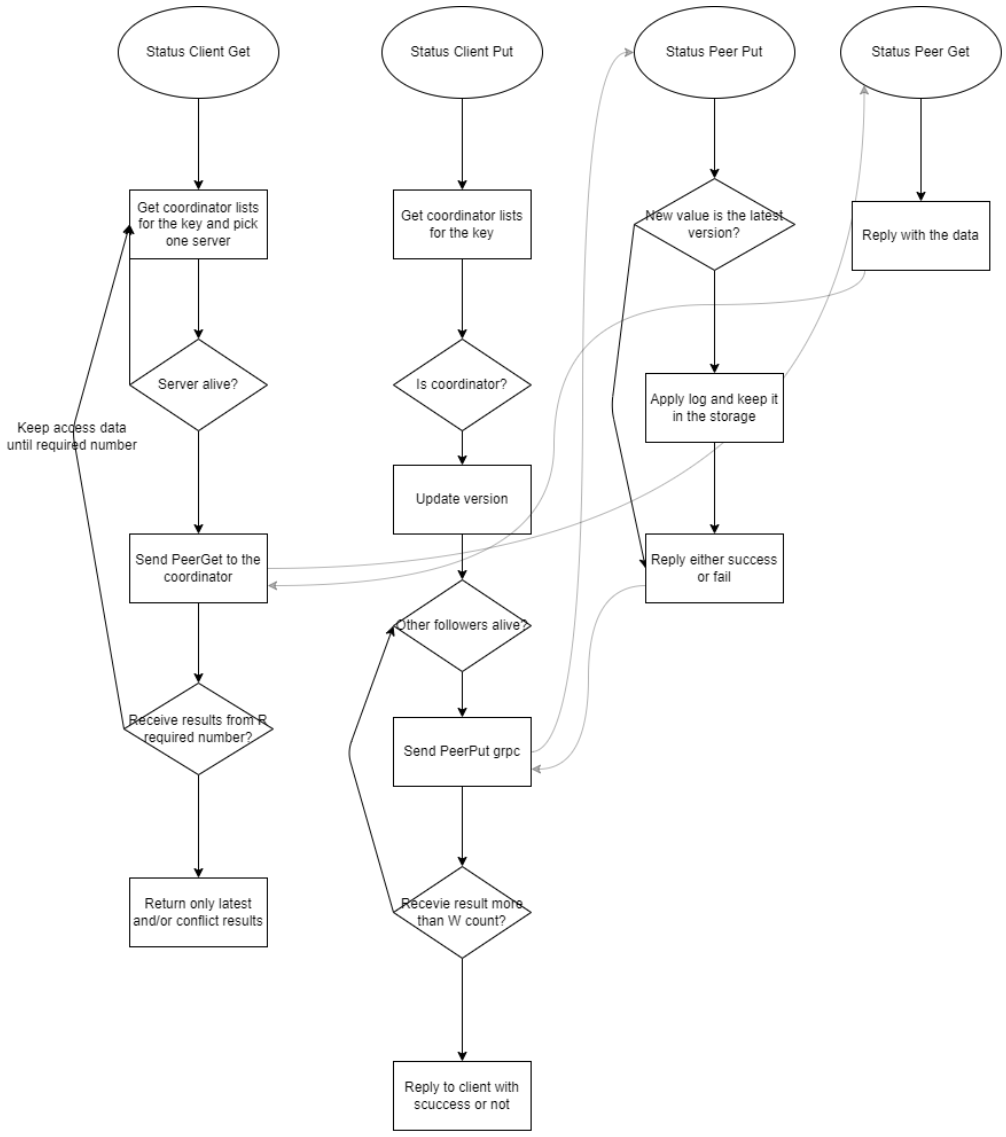

Figure 2: Server-side function workflow diagram

node maintains an updated record of the most recent interactions, which in turn contributes to the overall efficiency and robustness of the network.

1.5 Administrator library

The ”Admin” class provides the ability to send commands to the network nodes and monitor their status. It has the address of all nodes. There are three commands that Admin provides: Ping, JoinNetwork, and LeaveNetwork.

2 Correctness

2.1 Conflict record reconciliation

The purpose of the demonstration for reconciling conflict records is to test the functionality of the system. Initially, an initial key-value pair will be added to the system to ensure proper operation. Then, all servers will be shut down, a conflict record will be manually created, and the systems will be restarted. The modified test system will read the existing data from the disk and store it in memory. If a client wants to read or write the conflict record, they will have to decide which data to keep first before proceeding with their next steps.

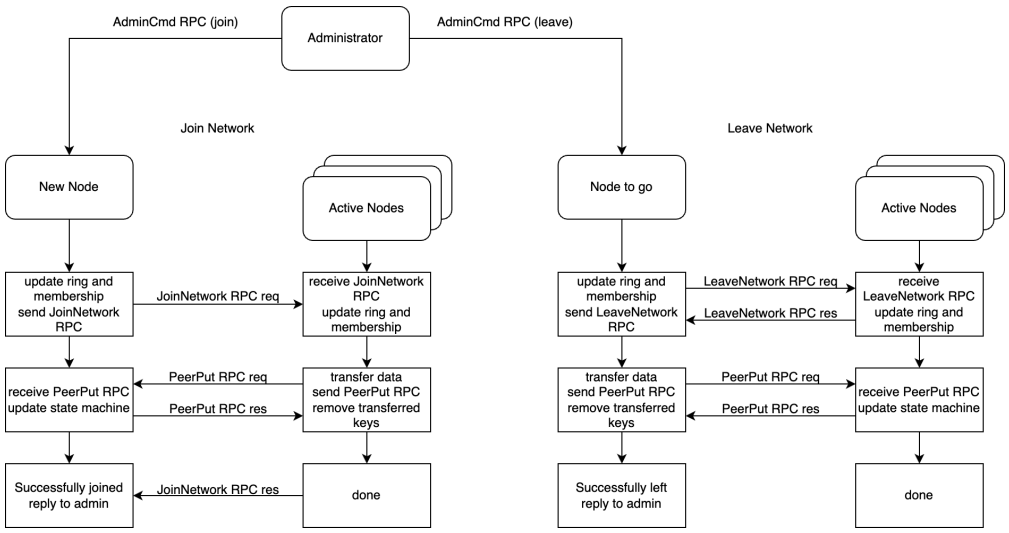

Figure 3: Membership change workflow diagram

2.2 Handle temporary failure

To demonstrate the system's ability to handle temporary failures, we will begin by starting the system with five servers. Initially, we will initialize a key-value pair. Next, we will shut down one of the servers that is responsible for that key and retrieve the data again. This demo will illustrate that even if a node experiences temporary failure, the system can still handle the data effectively.

2.3 Membership change

An administrator has the ability to issue commands to nodes, instructing them to either join or leave the network. Upon receiving a 'join network' command, the new node initiates a series of actions. First, it updates the consistent hashing ring. Then, it issues a 'join network' RPC to all other active nodes. This alerts these nodes to update their own hashing rings and prepare for data migration to the new node. Subsequently, all active nodes commence the process of transferring the relevant data to the newly joined node. Once this data transfer is completed, the new node is successfully integrated into the network.

The process for leaving the network is quite similar to joining it. However, in this case, the node that is set to leave must undertake an additional step before its departure. It must transfer its data to other active nodes, based on the updated hashing ring. This ensures data integrity and availability even after the node's removal from the network.

3 Performance Evaluation

3.1 Put and Get Performance

In this performance test report, we analyzed the put and get operations for two factors, the number of servers and the number of clients. Our system was designed to be configured with either 5 or 10 servers, and we tested the performance with either one or five clients sending requests to servers in parallel. Both put and get operations were performed synchronously for each client. Each client executes 100, 500, 1000, 1500, or 2000 operations.

The results showed that for put operations, overall, the system's performance was worse with more clients, as the system had to handle more key-value pairs (fig.4). This trend was evident from the circle and triangle data points. Additionally, we found that using more servers resulted in better performance. The difference in performance was more noticeable when comparing one client to five clients. One possible explanation for this trend is how we keep our state machine. In our current design, after we put the key into the servers, they store all key-value pairs from memory to disk. If we only use five servers, then more data needs to be kept per server compared to using 10 servers.

Regarding get operations, the overall performance did not have a significant difference (fig.6).. When the system includes more servers, the performance of get operations seem not to be improved, this may be due to the fact that we did not implement load balancing for tasks. Specifically, all clients communicated with a single server when getting data from the system in the current design, while they communicated with the different coordinators when putting data into the system.

After our presentation, we noticed something interesting about the latency of put operations. It didn't increase in a straightforward, linear way as we expected when we increased the number of put operations. We think there could be two reasons for this.

Firstly, the pesky network delay might be causing some trouble. As we send more and more put operations, the network traffic and communication overhead also go up, which can slow things down. Unfortunately, we couldn't set up the right experiment to really dig into this issue in our current setup.

Secondly, we're using a regular unordered map to manage our data, and that might be playing a part too. The speed of persisting data, like writing it to disk, may not increase linearly as we add more puts. Again, we couldn't fully investigate this in our current tests.

While these factors could explain the unexpected latency patterns, we'll need to do some more digging and experimenting to be sure. In future iterations, we're planning to run more thorough tests to really tackle these issues and understand how they affect our system.

Overall, our initial findings suggest that the latency of put operations can be a bit more complicated than we thought. By diving deeper and finding ways to optimize, we're determined to make our system faster and more predictable.

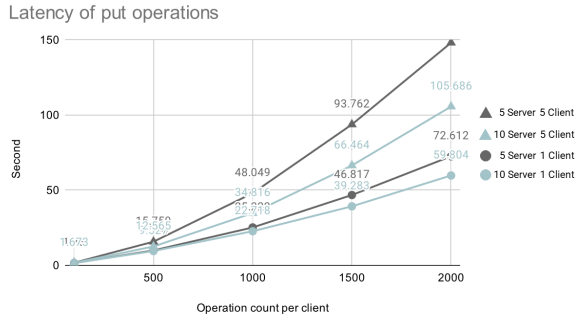

Figure 4: Latency of put operations

3.2 Optimized setting for read or write

We have optimized our performance by utilizing asynchronous RPCs. Initially, our system was set to wait for responses from three replicas, which proved to be an inefficient process. To mitigate this, we implemented multi-threading to facilitate asynchronous 'put' and 'get' operations.

As demonstrated in the accompanying diagram, asynchronous RPCs perform significantly better, particularly under heavy workloads. For instance, asynchronous RPCs are capable of executing 2000 operations within

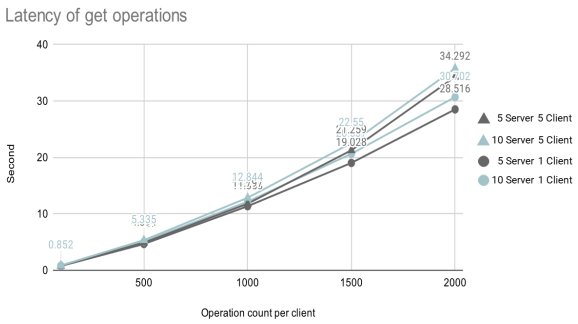

Figure 5: Latency of get operations

a span of 100 seconds. In contrast, synchronous RPCs require one hundred and fifty seconds to complete the same number of operations, illustrating the improved efficiency of asynchronous RPCs.

We also conducted tests with varying configurations for R (the number of nodes that need to agree on the read operation) and W (the number of nodes that need to agree on the write operation). Given that the number of read operations closely parallels the number of write operations and that the RPCs are executed asynchronously, we did not observe any significant differences in performance across these varied configurations.

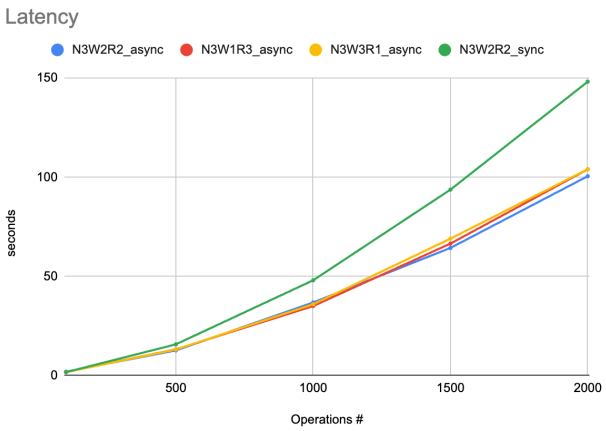

Figure 6: Latency of Async and Sync RPCs

References

[1] H. D. J. M. K. G. L. A. P. A. . . V. W. DeCandia, G. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, (6):205–220, 2007.