Distributed Database with Raft Protocol

Ruthvika Reddy Loka, Mahitha Pillodi, Mia Weaver

April 11, 2023

1. Introduction

Key-value stores have become increasingly valuable in the field of data management due to their scalability, speed, and flexibility. This type of database uses an associative array as its underlying data model, with each key linked to a single value, resulting in a key-value pair. The simplicity of this structure makes key-value stores an efficient solution for data storage and retrieval, particularly as the volume of data being processed increases.

Our goal is to build a fault-tolerant and highly available key-value store that can support PUT and GET operations, with both keys and values being strings of any size. To ensure strong consistency, we have designed our system to use Raft as the consensus protocol. We used gRPC for client-to-key-value store server calls, as well as internal Raft APIs like RequestVote and AppendEntries.

2. Methods

2.1. Replicas

The key-value store is typically implemented as a cluster of nodes that supports APIs, with a minimum of three nodes being recommended for fault tolerance. Ideally, the number of nodes should be odd, such as 3, 5, or 7. The Raft consensus algorithm is used to ensure replication and fault tolerance across the cluster nodes. Our distributed system is made up of 5 nodes hosted by Cloudlab and located in Salt Lake City, Utah.

Each replica is outfitted with a mechanism for communication with other replicas, participating in leader election, and appropriately handling incoming messages according to the RAFT protocol.

We ran trace route measurements between these nodes to observe the latency associated with network communication in our system. We note that node 2 has the lowest variation in communication round trip times compared to the other nodes, hinting at its centrality in the network. This is relevant to later results.

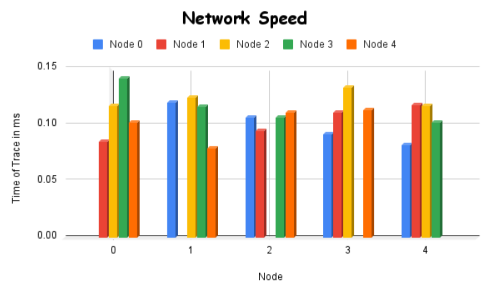

Figure 1: Trace route latency to replicas

2.2. Supported APIs

Our library provides support for two operations in the key-value store. These operations are as follows:
2.2.1. GET: The GET operation fetches the current value for the given key. It returns an empty string in case of a non-existent key.
2.2.2. PUT: The PUT operation re-writes the value for the given key. In case of a non-existent key, the PUT operation simply adds the key and value pair to the in-memory database.

2.3. Remote Procedure Calls

To facilitate message passing between the replicas, we use gRPC. The "raft.proto" file contains the Protocol Buffers definition for the communication protocol used by the Raft consensus algorithm in a replicated database system. We defined several remote procedure calls (RPCs) which can be used to interact with a Raft-distributed consensus algorithm.

The Raft service has the following RPCs:

- **VoteRequest:** Takes a message vote_4_me and returns a message voted_4_u. The RPC is used to request votes from other nodes to elect a leader.
- **AppendEntryRequest:** Takes a message appendEntry and returns a message appendEntry_pending. The RPC is used to append new entries to the log of the Raft algorithm.

- **heartbeatUpdate:** Takes a message heartbeat and returns a message heartbeat_response. The RPC is used to send heartbeats to other nodes in order to maintain leadership and detect failures.
- **getVal:** Takes a message getVal_request and returns a message getVal_response. The RPC is used to retrieve a value associated with a given key.
- **setVal:** Takes a message setVal_request and returns a message setVal_response. The RPC is used to set a value associated with a given key.
- **suspend:** Takes a message suspend_request and returns a message suspend_response. The RPC is used to temporarily suspend the Raft algorithm.

The "raft.proto" file also defines several message types that are used in the RPCs:

- **vote_4_me:** Used to request votes from other nodes during leader election.
- **voted_4_u:** Used to communicate the result of a vote to the requesting node.
- **appendEntry:** Used to append new entries to the log of the Raft algorithm.
- **appendEntry_pending:** Used to indicate the outcome of the append entry request.
- **heartbeat:** Used to send heartbeats to other nodes in order to maintain leadership and detect failures.
- **heartbeat_response:** Used to communicate the outcome of the heartbeat request.
- **commitVal_request:** Used to request committing a value associated with a given key.
- **commitVal_response:** Used to communicate the outcome of the commit value request.
- **suspend_request:** Used to request to suspend the Raft algorithm.
- **suspend_response:** Used to communicate the outcome of the suspend request.
- **getVal_request:** Used to request retrieving a value associated with a given key.
- **getVal_response:** Used to communicate the outcome of the get-value request.

All the message types contain several fields that are used to represent the data being passed between nodes in the distributed system.

2.4. Consensus: Raft Protocol

The Raft protocol entails that every server can assume one of three states, namely Leader, Candidate, or Follower, and each plays a vital role in achieving consensus. This report presents a detailed exposition of each state and its respective functions. Moreover, we provide an implementation of the primary components of the Raft Library, including Leader Election, Log Replication, and Log Persistence. Please find below an elaboration of these components.

2.5. Server States

The system operates with distinct states for each server. To maintain consistency, each of these server states is preserved as a constant variable with integral values. In our implementation, Follower is represented as "F", Candidate as "C", and Leader as "L".

Leader: This state is responsible for accepting client requests and ensuring that log replication is carried out at other servers. Generally, there is only one leader in the system during normal circumstances.

Candidate: This state is activated during the election process, wherein servers may request votes from other servers. A server that requests votes is known as a candidate.

Follower: This is a passive entity that doesn't issue requests of its own accord. Instead, it only responds to requests made by the leader or candidates. The Follower synchronizes its data with the leader and can initiate an election if the leader fails.

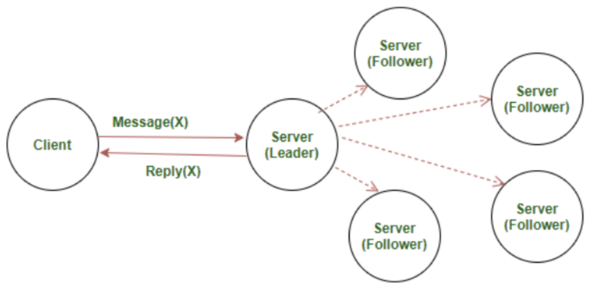

Figure 2: Server States

2.6. Leader Election

Leader election is invoked by one or more candidates upon session initialization. Each replica has a random delay value before it can declare itself a candidate and request votes. Once a replica has declared itself a candidate, it either waits until it receives a quorum of votes, or it receives a heartbeat message from the elected leader. In the former case, the replica updates its state to reflect its role as the leader. It begins to send heartbeats to the replicas and services database requests. In the latter case, it reverts its role to follower, and awaits further messages. Leader election is also invoked when the leader crashes or becomes too slow. Leader activity is monitored via heartbeats to followers. Each replica has a random length time window ranging from 150-300ms to receive a heartbeat from the leader before it invokes re-election. Upon re-election invocation, the candidate replica also updates its term before telling other replicas to vote for it.

2.7. Log Replication

Log replication is managed by the leader. Clients' PUT requests are handled by the leader, which consists of a command to be executed by the replicated state machines in the system. The leader first logs this locally and spreads the messages to all the followers about the updated entry. Followers log the updates in the message, and feedback to the leader as consent to the update. Once the majority consents to the update, the leader will then "commit" this transaction to its local state machine and respond to the client about the success. Messages to inform all the followers to commit their replicates are sent in the meantime. The "commit" actions to transact all the previous staged logs if any were left undone.

2.8. Database

We use a simple pickle file located on each replica to maintain the key-value store of the distributed database. Each replica's dispatcher code maintains a Python dictionary of pending database entries. Either a quorum of pending database entry acceptors will be reached and the value will be committed to the key-value store, or the leader will crash. In the latter case, it is up to the client to resend the put request to the new leader.

3. Design

In the Raft design, the client directs PUT requests to the leader node. The leader node propagates client entries to its followers. Once a majority of followers have accepted this value, it is committed to the database. We use a simple Python dictionary to interface with our database, which we persist to memory by writing to a pickle.

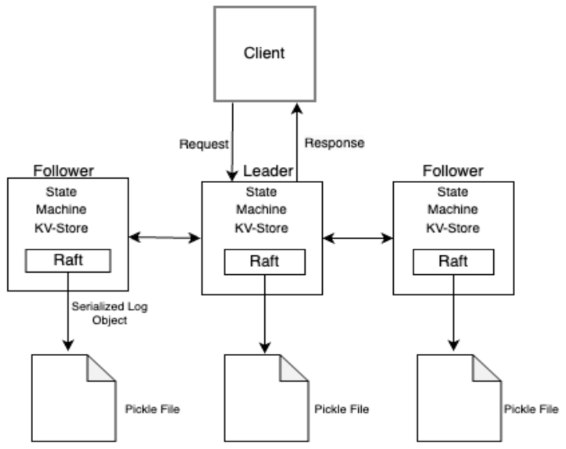

Figure 3: Overall Design

4. Evaluation

Several test cases were executed during the testing phase to evaluate the functionality of the system. The first test case focused on testing the leader election process and normal operations such as PUT and GET requests. The testing involved spawning all the servers and ensuring that only one leader was elected. Additionally, the responses to GET and PUT requests were evaluated to ensure they were in line with expectations.

The second test case focused on testing the system's ability to elect a new leader in the event of a crash and achieve a quorum. To achieve this, the current leader was deliberately crashed in a five-node cluster, and the status of each server was printed to confirm that another node took over leadership.

The third test case tested the scenario where the majority of nodes were down, and no leader was elected. This was achieved by crashing two nodes in a three-node cluster and verifying that the remaining node continued to be a candidate by printing its status.

Finally, the fourth test case evaluated the system's ability to suspend and replicate logs. The testing involved verifying that the client received a response after the logs were replicated to all the nodes. The logs on each server were also printed to confirm that log replication was occurring on all nodes. Additionally, the system's ability to recover from a crashed node was tested by suspending a node, sending requests to the other nodes, and checking whether the logs of the crashed node were synchronized with the majority of nodes after recovery.

5. Performance Evaluation

To evaluate the performance of the system, we consider several metrics. First is the time taken for a PUT request to be serviced by the leader node. We consider the latency between the invocation of the PUT service on the leader node and the subsequent commitment of the data on 1. the leader node and 2. the replica nodes. We collect these values for each node as it acts as a leader. We note that node 2 services commit requests faster than the other nodes, hence the investigation of the network referenced in *[2.1]*. This testing was done largely with small key, value inputs (strings of < 4 bytes) and achieves commit latencies of reliably < 900 ms.

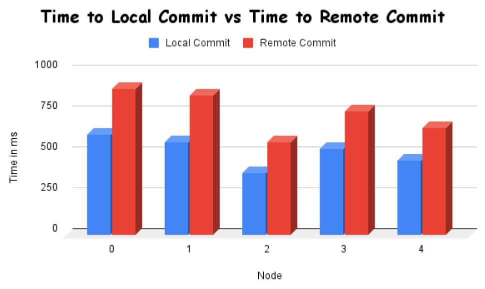

Figure 4: Time to local commit vs Time to remote commit

Next, we consider the time it takes for PUT requests with increasing value string sizes (with < 4-byte key string sizes) to be serviced by the leader node. We believe the latency observed for a string of 8 bytes is the result of some congestion somewhere in the network or an overload in the system, though, for consistency's sake, we took the first value observed for the test. Overall, the system provides commit latencies of reliably < 1600 ms for input strings of up to just over 4k byte strings.

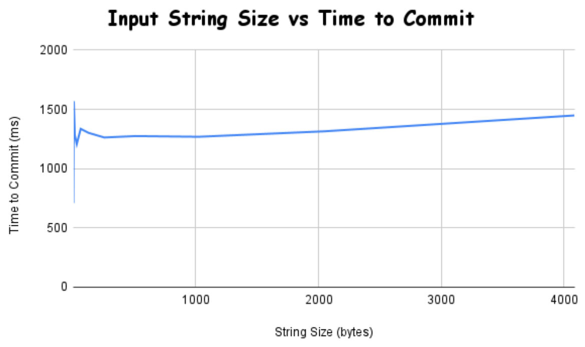

Figure 5: Value of PUT request size in bytes vs. time to commit on all replicas

Finally, we consider the time it takes for the system to elect or re-elect a node upon system invocation and leader time-out. We consider this on two axes. First is the latency between election invocation and the process by which a candidate node detects it has *won* the election and declares itself a leader. Second is the latency between election invocation and the process by which a candidate node detects it has *lost* the election and declares itself a follower. We noticed no significant difference in latencies incurred by the initial election vs. subsequent re-elections within a single raft session. However, many raft sessions in a row seem to degrade system performance such that the nodes need to be rebooted to resolve the issue. In this test, node 4 observed a disproportionately high latency when detecting a quorum of votes reached. This node was tested last after a series of raft sessions, and thus this result could have been due to network congestion or system overload.

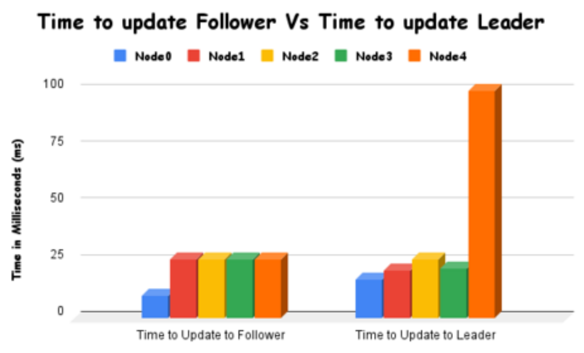

Figure 6: Time to update in case of election failure across all nodes vs. time to update in case of election success across all nodes

6. Discussion

During the implementation of our project, we encountered several challenges, particularly in attempting to modularize raft, and in integrating gRPC and raft. We realized that developing gRPC and raft in parallel would have been a more effective approach than creating two separate code bases. We also started with an attempt at a more modularized implementation of the raft protocol, such that an event loop in our server code would call upon a client module, leader module, or follower module depending on its state. This was incredibly difficult to coordinate with the need for a global state that is accessible by the process in which the event loop is executed *and* connected clients and replicas as they invoke gRPC mechanisms. We perhaps wasted some time with this first implementation attempt, though it ultimately helped us better understand the raft protocol. Additionally, we faced difficulty in achieving simultaneous communication between all nodes. However, we found a solution by utilizing the "sleep" function.

On the day of our project demo, we faced a critical issue when our code suddenly stopped working. Specifically, our threads were hanging and gRPC calls were not being made. After a brief moment of panic, we decided to reboot our nodes on the cloud lab, which allowed us to revert our code back to its previous state. This experience highlighted the importance of being able to quickly troubleshoot and resolve issues during the development process.

Overall, this project was fun and challenging. This project helped us to greater appreciate some of the insights discovered in some of the papers we have read this semester.

7. References

- Referenced this YT tutorial for general grpc tutorial (very helpful by the way)
- Referenced this repo for an idea of how to use gRPC for raft; looked at at their implementation of setVal and getVal in raft.proto and client.py specifically.