KVRaft, a Distributed Key Value Storage System Using Raft

CPSC 416 Final Report

Cindy Miao, David Wu, David Yang, Eric Yan, Michelle Gu, Parth Garg

1. INTRODUCTION/MOTIVATION

In distributed systems design, consensus is the fundamental problem that limits a set of servers' ability to survive faulty processes. At its core, consensus involves the coordination and agreement on the same values by multiple servers, where any agreement reached is final.

Our team's motivation for tackling the consensus problem stems from our desire to address data safety in distributed settings without creating a slow and potentially blocking system. Challenges like network failures, node failures, byzantine failures, heterogeneous systems/environments, and even just the simple coordination of multiple concurrent processes makes consensus a challenging but rewarding problem. Hence, our team decided to recreate the Raft protocol with a coordinator node. Through this experience, we plan to attain greater understanding of consensus and distributed systems in general.

To demonstrate the capability of our simplified Raft implementation, we plan to apply Raft to a key value transaction storage system called **KVRaft**. In **KVRaft**, there is a known number of servers and an arbitrary number of clients. The client is permitted to issue only two types of operations: puts to update a key's value and gets to retrieve a key's value. The client is also only able to issue these operations to the lead server. The servers are replicated state machines that will all contain similar logs and the same persistent data. Once some logs are committed, the system ensures that the cluster's majority has the same committed value.

2. DESIGN

As our focus is on realizing the underlying Raft algorithm, we plan to borrow parts of the implementation of our key value storage system from assignment 3 to this project. Specifically, we envision that we will be using similar code and structure for the client and coord components. To make implementation feasible within an one-month timeframe, we also decided to add the following simplifying assumptions

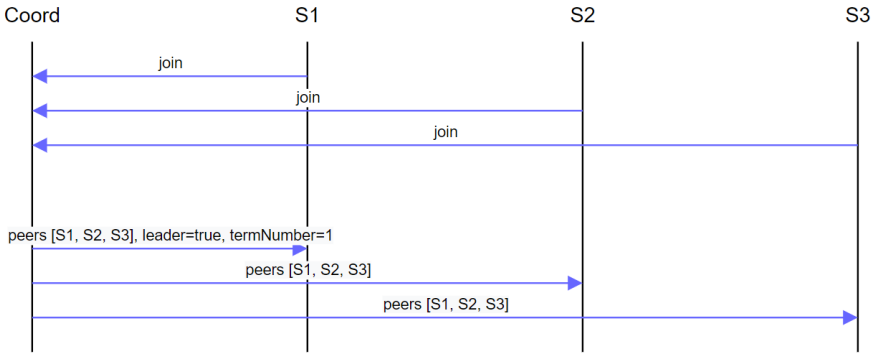
Assumptions

- The initial number of servers is 2N+1, where $1 \leq N \leq 3$, and the max number of servers is 7
- Servers are allowed to join only during startup, rejoins not permitted for failed servers
- No server can crash during the joining phase
- Coordinator (coord) node exists and does not fail (see its role in the next section)
- During a network partition, the coord is assumed to be a part of the partition
- There are no Byzantine failures
- There can be no more than 256 clients and each client can issue no more than 2^32 ops

2.1 Coordinator

Join process

The coord is used to maintain the topology. Servers are admitted into the cluster by issuing a join request to the coord. The coord is always aware of the max number of servers allowed. So, join requests exceeding the max capacity are rejected.

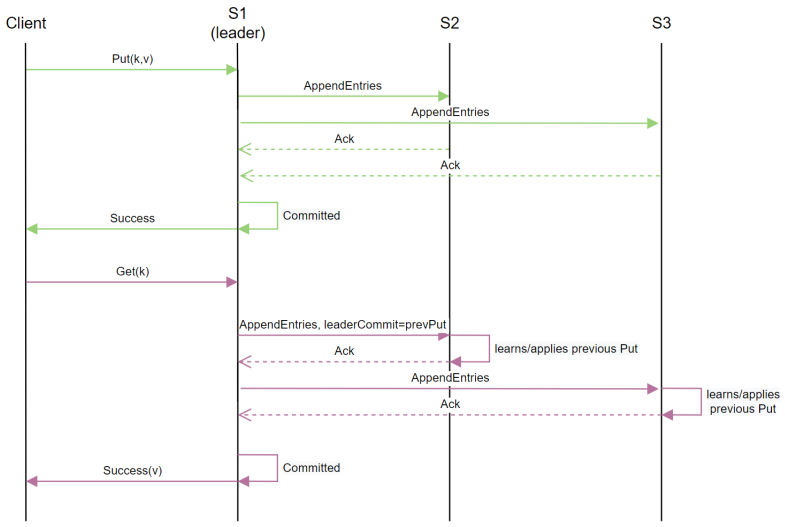


Client-Coord

The client is unaware of the leader during startup. It contacts the coord to find the leader. The coord only provides the leader once the server join process is completed (number of servers joined equals the number of servers expected to join in config) and a leader has been selected. If a leader fails, the client finds out about the new leader via the coord.

2.2 Raft and Log replication

After a leader is selected, it starts accepting client requests. Once a leader receives a command from the client, it **1)** appends the entry to its log, then **2)** sends *AppendEntries* RPCs in parallel to each of the other servers to replicate. This call should be retried indefinitely until all servers store all log entries. Upon receiving acks from a majority of peers (including leader) that the entry is replicated, the command is "committed".



Committed entries guaranteed to eventually be executed by all of the available state machines. To implement this, the leader tracks highest committed index, and includes it in future AppendEntries RPCs/heartbeats. When a follower learns that an entry is committed, it then applies the entry to its local state.

Maintaining Log Consistency

The leaders' log and the follower's log may be diverged (due to node failure, network partition, etc). To handle inconsistency, the leader forces the follower's log to copy its own. To do this, the leader includes the previous log entry's index (*nextIndex*) and term in *AppendEntries*. The follower should check to see

if there is the same index and term entry in its log before it adds the new entries to its log. Else, it should reject, and the leader will decrement *nextIndex* and retry the AppendEntries RPC until the call succeeds, in which it finds the last similar entry on the follower's log, and adds the new entries from the leader's log.

2.3 Client interactions

The kvslib Get and Put API is very similar to that of A3, except the client interacts only with the leader. Initially the client requests and waits for the leader's address from the coord (coord designates the leader). If the leader crashes while executing a client request, the coord should notify the client about the failure and the new leader server (when it's selected), and the client will retry the request with the new leader. The goal is linearizability: the system behaves as if each operation is executed exactly once, atomically sometime between sending of the request and receipt of the response. In addition, the client generates a unique Id for each request (OpId), and when retrying the client reuses the original Id. The servers need to track the latest Id for each client, along with the associated response. When the leader detects a duplicate, it sends the old response without re-executing the request.

2.4 Failures

Server-Coord

Failure detection and the leader selection protocol are centralized into the coord.This way, the servers are only required to ensure that the data is properly replicated and that the consistency semantics (guarantee data consistency for server failures and network partition scenarios) are met.

The coord uses the fcheck library to monitor the status of all the nodes. If a leader is determined to be offline, the coord finds a most up-to-date node by comparing the log index and term of the last entry in the log file of each node, and then proceeds to notify the node about the change in leadership.

The raft algorithm requires a majority of the servers to remain online. So, if the coord detects N/2 + 1 failures, the kvs does not stay functional any longer.

Leader Failures

From the diagram in log replication, we see that the leader may fail after it commits an entry, but followers have yet to learn about this commit. Thus, we must ensure that the new leader has all the log entries committed in previous terms, (see Leader Completeness Property in Raft). To ensure this, coord will compare servers' logs, and choose the one with the most up-to-date log, and committed means the entry in current term is replicated on the majority, with the preceding entries also committed.

3. IMPLEMENTATION

Our implementation is split into 3 sections: server, coordinator, and client.

3.1 Server

Raft Roles

Following the raft algorithm, a cluster of servers will contain 1 leader server, with the rest being followers. Thus, a server can either have a leader role or a follower role at a time, so either

raftLeaderLoop or **raftFollowerLoop** will run depending on the state of the server. Roles are assigned on startup by coord, and may be changed during failure when a new leader is designated by coord. Note that the candidate role is not needed, as our design uses the coordinator to assign leaders.

Key data structures and state

Log. This is a list of entries, with each entry containing an index, term, and the command to apply to the state machine. The term is the leader's term when receiving the command. The command includes a *type*, which is either get, put, or no-op (described later in handling duplicates), and *ClientInfo*, containing ClientId and OpId. This list is 1-indexed.

Key-Value Store. The state of the state machine is stored in a map of keys to values.

commitIndex. The highest log entry known to be committed. Once the leader receives a majority of acks from followers for an entry, it will increment commitIndex, which signifies the entry as committed. This is updated on the followers when the leader sends an AppendEntries with its updated commitIndex.

lastApplied. The highest log entry applied to the kv store. If *commitIndex > lastApplied,* the server will apply all log entries up to commitIndex to its kv store.

Log replication

The leader server will receive **Get** or **Put** RPC calls from the client. For both **Get** and Put requests, we first check if the request is a duplicate (see details in next section). If the request is not a duplicate, then the request is converted into an internal ***ClientCommand*** object that will be sent via a channel into the leader's server loop. This operation ensures that requests are serialized in order of arrival. The **Get** or **Put** RPC calls will block until the request is safely replicated across a majority of servers. This step is done by sending **AppendEntries** to all follower servers in parallel in **leaderHandleCommand**.

Upon receiving **AppendEntries** from the lead server, each follower will

- Check if the request satisfies the raft checks to maintain log consistency, detailed in our code.
- Append the entries to its log, and reply success to the leader.
- Update its `commitIndex` to the `LeaderCommit` index sent by the leader in this call as it learns about newly committed entries by the leader. Once `commitIndex` updated, the follower will apply the newly committed log entries to its kv state, done by **doCommi**t.

Checking for duplicates

In the case of some node or network errors, servers may receive duplicate requests from client retries. To detect these duplicate requests, extra client metadata is stored in the state machine (***lastClientLogEntry***, ***lastClientKVCommand***). Upon seeing duplicated requests, the server will either immediately respond with the correct results of the duplicated requests if it has been committed or let the client retry again after some time to wait for the server to commit the request. The client metadata is updated and propagated the same way as normal request data (with **AppendEntries**).

3.2 Coordinator

Join Process

- Upon server start, it makes a **RequestServerJoin** rpc call to the coordinator.

- The coordinator queues the requests until it receives N such requests, and selects the server with ID 1 as the leader, the rest as the followers.
- The coordinator makes a **joinProtocolNotifyServer** rpc call to all the servers passing along the information contained in the **JoinResponse** struct`{ServerId uint8, Leader bool, Peers []ServerInfo, Term uint32 }`

Leader Discovery - Client

- At client startup, it makes a **GetLeaderNode** rpc call to the coordinator. Once the join process is completed, the coordinator returns the leader information.

Failure Detection

- The Coord uses the fcheck library to monitor all the servers, and maintains a cluster view that's updated during server failures. If fcheck detects a leader failure, it runs the leader selection protocol. If a network partition disappears, the servers that are now reachable by the coordinator and included back into the cluster view.

Leader Selection

- The coordinator selects the most up-to-date leader by first acquiring the **ServerLogState** `{ServerId uint8, Term uint32, LogIdx uint64, Token tracing.TracingToken}` of each available server. The coordinator acquires this information by making a **GetLogState** rpc call to each server.
- The coordinator completes the leader selection protocol by making a **NotifyFailOverLeader** rpc call to the new leader, specifying the new term number of the system.
- The coordinator notifies the client about the change in leadership by making a **ChangeLeaderNode** rpc call to the client, specifying the server address of the leader.

3.3 Client

The client code adopted the A3 implementation from one of our A3 groups, and modifications were made based on it to make it work with the project. A client interacts with the server chain through kvslib, which only interacts with the leader of the server chain.

Start

- **KvslibStart{clientId}** only is the first *ktrace* recorded by a client, signaling the start of the client's kvslib
- After the kvslib establishes a RPC connection with the coord, the client records **LeaderReq{clientId}** in *ktrace* immediately before sending a RPC request to coord to get the address of the current leader node.
    - Coord only returns a response after the server chain is formed and a leader is selected, at which point the kvslib records **LeaderResRecvd{clientId, serverId}** in *ktrace*
    - The kvslib uses leader information it has received from coord to set up RPC connection with the leader
- Before *Start* returns, it initiates 2 KVS functions as goroutines: *handleFailure()* to handle leader failure when receiving a RPC call notification from coord, and *sender()* to send Get and Put requests to leader

Leader Failure

- Coord notifies kvslib about leader failure by sending a RPC call *ChangeLeaderNode()*, which unblocks *handleFailure()*, killing the current connection to the failed leader
- **LeaderReq{clientId}** is once in *ktrace* for every failure notification the kvslib receives from coord. This is recorded right before the client asks coord for the new leader
    - **LeaderResRecvd{clientId, serverId}** is recorded in *ktrace* immediately after kvslib receives the new server from coord, and a RPC connection with new leader is set up.

Put Request

- A new *putTrace* is generated for each new put request
- **Put{clientId, opId, key, value}** is recorded in *putTrace* before kvslib sends a Put request to the leader for the first time.
    - If the RPC call returns with an error, or if the leader is changed, the same request is sent again, but this time, not recorded in *putTrace*
    - The resend process ends when the any of the RPC calls for the same request returns a successful response, at which point the kvslib records **PutResultRecvd{opId, key}** in *putTrace*

Get Request

- A new *getTrace* is generated for each new put request
- **Get{clientId, opId, key}** is recorded in*getTrace* before kvslib sends a Put request to the leader for the first time.
    - If the RPC call returns with an error, or if the leader is changed, the same request is sent again, but this time, not recorded in *getTrace*
    - The resend process ends when the any of the RPC calls for the same request returns a successful response, at which point the kvslib records **GetResultRecvd{opId, key, value}** in *getTrace*

Stop

- The kvslib closes all opened connection that can be accessed by the KVS struct
- **KvslibStop{clientId}** is recorded in *ktrace* and is the last trace recorded by the kvslib

4. EVALUATION

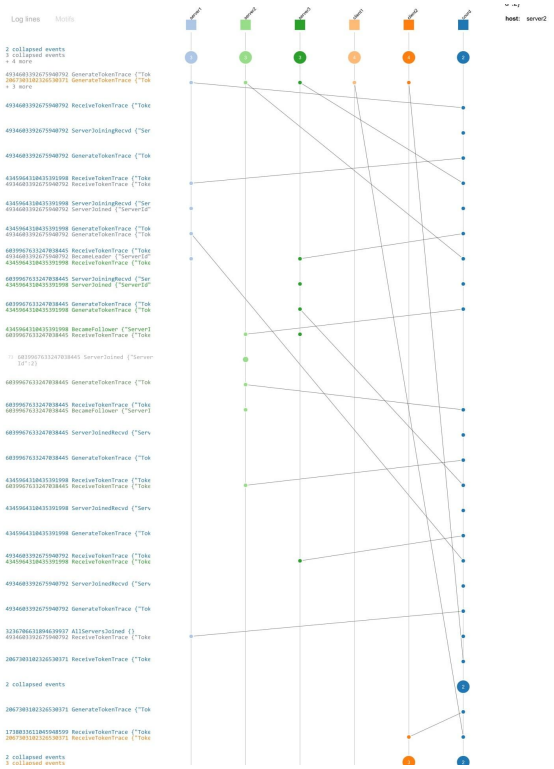
Unit testing

We used unit testing (in addition to running the system) to help us verify correctness. Unit tests can be run by starting the system as normal, with tracing server, coord, and 3 servers, and then running `cd raft && go test.` The unit tests verify normal get/put operation, and more importantly, a network partition scenario, where the leader (node 1) gets isolated from the rest of the cluster.

Tracing

Using the tracing library and ShiViz, we were able to create and visualize traces of our system under various scenarios. The traces allowed us to observe that key events and function/RPC invocations are happening in the correct order. By observing the traces, we are also able to verify that the strong consistency guarantees are met. For example, the ShiViz diagram below shows the normal execution

of the system with 3 servers and 2 clients. The left diagram mainly shows server and client interacting with the coord to respectively join the system and find the leader server. The right diagram shows the normal operation of Get and Puts issued by the client and replicated by the leader server. We also created traces for various failure scenarios which can be found here.



Performance Testing

To evaluate the performance of KVRaft, we plan to run a number of experiments. The metrics we will use include the average running time to perform a set of read only tasks, write only tasks and mixed read/write tasks. The tests will be conducted in different scenarios with changes to the number of clients and servers. The tests will also be run for a single server and for a chain replication key value system (assignment 3) so that we can compare KVRaft's relative performance. In particular, the single server case will be our baseline.

- We tested first by comparing A3 and our KVRaft with 3 servers for each. We ran two test cases: 100 puts and 100 gets. A3 had around 1.5-2 seconds for both while KVRaft had 2-2.8 seconds for both. Similar numbers(in reference to both types of replications) were outputted when we ran 100 alternating puts and gets. We did this by adding the "time" identifier before "go run cmd/client/main.go client_id" which printed out the runtime for the commands in the main files to finish executing.
- The shorter execution time for 100 Gets from A3 is potentially due to Gets being sent to the tail server in A3, and this shortens the time needed to receive a response from the server, as Get requests do not need to go through the entire chain of servers. For the Puts, our hypothesis is that our implementation in A3 of replicating the KV down the chain is faster for smaller server numbers than our implementation in the KVRaft, likely due to difference in uses of buffering or goroutines between the two projects.

- When we increase the number of server counts(3 to 5), the chain replication from A3 still performs better in terms of puts and gets. With A3 getting 1 second for gets 3 seconds for 100 puts, and kvraft getting 6 seconds for both. An explanation for this is that the logging updates and raft rules make KVRaft slower than chain replication for our version of A3 we used to measure performance with.

5. DEMO PLAN

Environment information: all nodes will be running on CS student servers.

- Normal operation (no failures/joins)
    - Base case: There will be 1 client and 3 server nodes. The client will send a few requests to the KV Raft service, and we will use ShiViz live to demonstrate correctness.
    - Complex case: There will be multiple clients, sending many requests to the KV Raft service, and all the requests should be successfully replicated to the system.
- Survive at least 3 node failures:
    - To survive 3 node failures, we will have 7 nodes to start in the cluster. We will then fail both a leader node and a non-leader node via terminal (ctrl+C), and show that the service is operating as normal.
- Demonstrate system can join and utilize at least 3 new nodes:
    - Upon service startup, we will show the join process for a cluster of 3 nodes. All of the nodes should receive a list of nodes in the cluster, one of the nodes should become a leader, while the others will become followers, and the kv raft service should be ready to receive client requests.

6. CONCLUSION

From this project, we were able to explore a new family of consensus algorithms outside of the classic Paxos algorithm. Through reading about Raft and coming up with our own design that featured a coordinator, we attained deeper understanding of core concepts in distributed systems, including consensus, safety, liveness and consistency. The end product satisfied our evaluation criteria and was interesting to build in general. Nevertheless, there are still many optimizations and architectural improvements that we did not add in this submission. We look forward to future opportunities to improve KVRaft and make it more suitable for practical systems.

7. REFERENCES

- https://medium.com/the-sixt-india-blog/raft-and-paxos-a-brief-introduction-to-the-basic-consensus-protocols-powering-distributed-systems-1a0ef7ca3acb
- https://web.stanford.edu/~ouster/cgi-bin/cs190-winter20/lecture.php?topic=raft
- https://raft.github.io/
- Raft paper: https://web.stanford.edu/~ouster/cgi-bin/papers/raft-atc14