Towards Understanding an "Understandable" Consensus Algorithm

Edward Jin, Henry Lin, Dhylan Patel

1. Background & Motivation

Consensus remains one of the most challenging and fundamental problems in the field of distributed systems. In distributed environments, where nodes can fail at any time and messages can be delayed or lost, reaching agreement among all participating nodes is particularly difficult. To tackle these challenges, Leslie Lamport introduced the Paxos algorithm, a method for achieving consensus in the presence of failures [2]. Unfortunately, Paxos is widely regarded as one of the most difficult algorithms to understand and implement correctly. Its complexity often becomes a barrier to practical adoption, particularly for developers or students new to distributed systems.

To address the steep learning curve and implementation challenges posed by Paxos, the Raft consensus algorithm was proposed. Raft was designed with clarity and ease of comprehension as first-class objectives, in contrast to Paxos, which often appears opaque or unintuitive to many learners [1]. This claim of improved understandability is supported by an empirical user study conducted by the authors, in which students who were taught both Raft and Paxos were able to answer questions about Raft with higher accuracy and confidence [1]. As students currently in a distributed systems course, we were inspired to explore this claim ourselves by attempting to understand and implement Raft from scratch. Our repository can be found here: github.com/EddieJ03/223b-raft.

2. Goals & Success Criteria

The goals of our project are:

1. To develop a correct and working implementation of the Raft consensus protocol using Go and gRPC.
2. To write a set of tests that verify the correctness of our implementation (note: we will not use stress tests).

Our implementation will concentrate on the three core components of the Raft protocol:

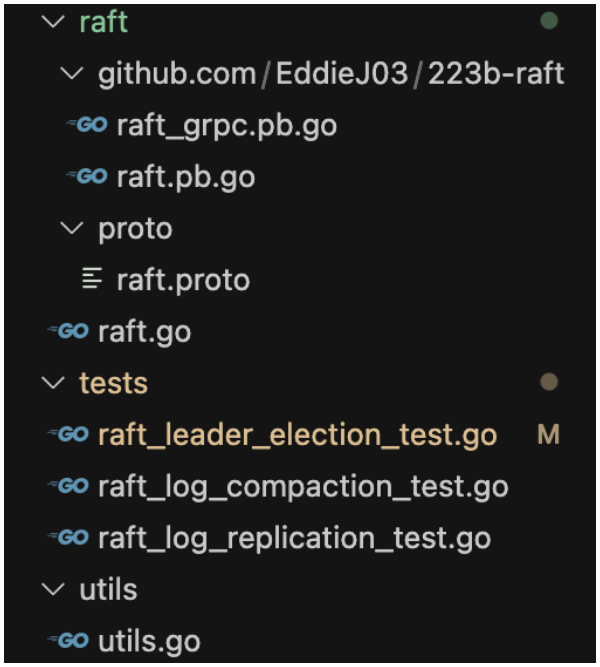
- **Leader Election**
- **Log Replication**
- **Log Compaction**, using the snapshotting technique described in the Raft paper [1].

We will rely heavily on the original Raft paper [1] as our primary reference, which provides extensive details to guide our implementation of these components. For this project, we define

success as the development of a Raft implementation that passes the tests we design. As a final note, it is worth mentioning that we do not plan to develop a client interface for our Raft protocol implementation in this project. Our focus will be exclusively on the internal mechanics and correctness of the Raft protocol itself.

3. Implementation Design

Our codebase is structured like the following (not all files shown):



For our Raft implementation, the most important files are **raft.go**, **raft.proto**, and **utils.go**.

The raft.go file contains the structs needed and methods & functions necessary to run the core Raft logic. An important struct defined in raft.go is the **RaftNode** struct. Here, we outline some of the important fields in this struct:

- **Path** – Filesystem path used for persistent storage for this node
- **Mu** – General mutex for protecting concurrent access to the node state
- **Id** – Unique identifier for this node
- **State** – Current role of the node (Follower, Candidate, or Leader)
- **CurrentTerm** – The latest term this node has seen
- **VotedFor** – Node Id that this node voted for in the current term
- **peers** – Map of peer node Ids to their network addresses
- **electionReset** – Timestamp of the last election timeout reset
- **Shutdown** – Channel used to signal shutdown of the Raft node
- **Logs** – Persistent log entries used in Raft's replicated log
- **CommitIndex** – Index of the highest log entry known to be committed

- **lastApplied** – Index of the highest log entry that has been applied to the state machine
- **StateMachine** – In-memory key-value store representing the node state
- **leaderId** – Id of the current known leader in the cluster
- **Snapshot** – Current snapshot of the state machine used to truncate logs and save space
- **leaderNextIndex** – For leaders: the next log index to send to each follower
- **leaderMatchIndex** – For leaders: the highest log index known to be replicated on each follower

The operations our implementation of Raft will store in its logs are also defined in raft.go, and it looks like the following:

type **Log** struct {
    Term int32
    Op int32
    Key string
    Value string
    Index int32
}

The term and index fields are both integers. Operation can be one of 0 for Set, 1 for Delete, or 2 for NoOp. Key and Value will be of type string.

The utils.go file contains some helper functions. The most important one here used for testing is ServeBackend. This ServeBackend function starts running a gRPC server registered to run RaftNode logic. The above-mentioned RaftNode struct is defined externally and then passed into ServeBackend used to register the gRPC server. We chose this setup because this makes it possible for us to directly access the fields of a RaftNode while testing. The ServeBackend function also takes in a shutdown channel that enables shutting down a RaftNode (and all associated resources) externally from ServeBackend.

Finally, the raft.proto file defines all the RPC calls we use. The defined services match what is outlined in the Raft paper [1]. In particular, here are our three services:

- rpc **AppendEntries**(AppendEntriesRequest) returns (AppendEntriesResponse);
- rpc **RequestVote**(RequestVoteRequest) returns (RequestVoteResponse);
- rpc **InstallSnapshot**(InstallSnapshotRequest) returns (InstallSnapshotResponse);

The AppendEntriesRequest has the following fields:

- term: Leader's current term
- leader_id: ID of the leader sending the request
- prev_log_index: Index of the log entry to precede new ones in entries field
- prev_log_term: Term of the log entry to precede new ones in entries field
- entries: array of Entries to store (empty for heartbeat)

- leader_commit: Leader's commit index

The AppendEntriesResponse just contains the term of the Node that replied and a boolean to indicate whether the AppendEntriesRequest succeeded.

Each Entry in the array of Entries sent by a leader to a follower has the same fields as the Log struct defined above.

The RequestVoteRequest has the following fields:
- term: Candidate's current term
- candidate_id: Candidate requesting vote
- last_log_index: Index of candidate's last log entry
- last_log_term: Term of candidate's last log entry

The RequestVoteResponse just contains the term of the Node that replied and a boolean to indicate whether the vote is granted.

The InstallSnapshotRequest has the following fields:
- term: Leader's current term
- leader_id: ID of the leader sending the snapshot
- last_included_index: Index of last entry included in the snapshot
- last_included_term: Term of last entry included in the snapshot
- data: Serialized state machine snapshot data (key-value map)

The InstallSnapshotResponse just contains the term of the responding Node.

4. Testing

For each of the three Raft components, we have tests falling under one or more of the following situations, where correct behavior is defined as different nodes never reaching an inconsistent state and no data loss when we have a majority of nodes alive:

- All nodes are operational
- Leader goes down or a follower goes down
- Minority of nodes go down
- Majority of nodes go down
- Node joins

All tests are to be run on a local machine. Running the set of tests for each part could take 1-2 minutes. Now we will walk through the testing suite we have developed for each component.

4-1. Leader Election Tests:

As previously mentioned, we have a Raft struct that holds each node's properties, and the ServeBackend function that sets up the gRPC server to process incoming requests. For all tests in this section, the election timeout values by default range from ~1.5 to ~2.5 seconds, and heartbeats roughly get sent every ~0.5 seconds. Finally, we have decided to set each RPC call timeout to 1 second.

To assist us with checking when the leader election has stabilized, we introduce a helper function called **waitForStableLeader**. We repeatedly get status updates from all nodes through a channel, which receives updates approximately every 1/10th of a second, for a set amount of time. Each time we receive an update, we count how many leaders there are. If exactly one leader is present and it's the same one we saw last time, we increment a stability counter. If we see the same leader three times in a row, we assume the leader is stable and return its ID. If the leader changes or we see no leader (or multiple), we reset the counter. If time runs out, we give up and return -1.

We wrote **6 tests** to test leader election correctness:

1. TestLeaderElection: Tests that a three-node cluster successfully elects a leader within a reasonable time frame. We do this by calling the aforementioned waitForStableLeader function with a wait of roughly ~10 seconds.
2. TestLeaderFailElection: Tests that the system recovers by electing a new leader when the current one fails. Again, we do this by calling the aforementioned waitForStableLeader function with a wait of roughly ~10 seconds.
3. TestJoinCluster: Tests that a newly added node joins the cluster as a follower and does not immediately start a new election. Instead of calling waitForStableLeader and waiting for ~10 seconds, we shorten it to roughly ~3 seconds (to account for node setup time and receiving heartbeat).
4. TestNoLeaderWithMinorityNodes: Tests that no leader is elected without a majority. This test intentionally only starts up one Raft node out of 3. We can't wait indefinitely, so we call waitForStableLeader with a timeout of 20 seconds, and see if no leader is elected within this time.
5. TestFollowerFailure: Tests that no election occurs if a single follower crashes. After crashing the follower, we check for a stable leader within roughly ~3 seconds.
6. TestLeaderAndFollowerFailElection: Tests that at first we can get a stable leader within ~10 seconds, but once we crash the original leader and one follower, no election can be reached. Again, we call waitForStableLeader with a timeout of 20 seconds, and see if no leader is elected within this time.

4-2. Log Replication Tests:

For all tests in this section, the election timeout values by default range from ~1.5 to ~2.5 seconds, and heartbeats roughly get sent every ~0.5 seconds. Same as the Leader Election Tests, we use the ServeBackend function to start Raft nodes and the corresponding gRPC servers. For

all tests, after electing a leader, we call **waitForStableLeader** with a timeout duration of 10 seconds.

To assist us with checking for replication, we defined two helper functions: **waitForCommitIndex** and **waitForLogReplication**.

The **waitForCommitIndex** function monitors a group of Raft nodes and waits within a specified timeout period until all non-nil nodes have a commitIndex that matches the specified expected commit index. It does this by checking each node's commit index every 1/10th of a second using a ticker. If at any point all nodes match the expected commit index, it returns true. If the overall specified timeout passes before this condition is met, it returns false, indicating that consensus on the commit index was not reached within the timeout period.

The **waitForLogReplication** function ensures that all non-nil Raft nodes have logs of a specified length and that the logs across nodes are identical, within a specified timeout period. It periodically (every 1/10th of a second) selects the first non-nil node as a reference, ensures it has the specified length, and checks that all other nodes have logs of the same length and that each entry in the log exactly matches the corresponding entry in the reference log. If all nodes match the reference, it returns true. Otherwise, when the timeout is reached, it returns false.

We wrote **8 tests** to test our log replication correctness:

1. TestOneOperationLogReplicationNoFailures: Tests log replication with a single operation (Set "key1" to "value1") across three nodes, and we do not crash any node. We have waitForCommitIndex and waitForLogReplication timeouts set to 5 seconds, so we expect each node to achieve the appropriate commit index and log entries within 5 seconds. Finally, we verify the nodes' states are as expected.
2. TestMultipleOperationsLogReplicationNoFailures: Tests log replication with multiple operations (5 operations, a mix of Set and Delete commands) across three nodes, and we do not crash any node. We have waitForCommitIndex and waitForLogReplication timeouts set to 5 seconds. Finally, we verify the nodes' states are as expected.
3. TestLogReplicationSingleFollowerFailureThenRecovery: Tests log replication when one follower crashes after some initial operations, more requests are submitted to the remaining nodes, then the follower rejoins the cluster. We verify that the crashed follower catches up when it recovers by calling waitForCommitIndex and waitForLogReplication with timeouts set to 5 seconds. Finally, we verify the nodes' states are as expected.
4. TestLogReplicationMinorityAlive: Tests that operations cannot be committed when only a minority of nodes (just the leader) remain alive. We have waitForCommitIndex with a timeout of 15 seconds, using the large amount of time that has passed as a proxy for no progress being made on the commit index.Finally, we verify the nodes' states are as expected.

5. TestLogReplicationMultipleFollowerFailAndVariableRecover: Tests various scenarios with 5 servers where 2 crash and a **different** amount recovers (0, 1, or 2). For this test, we use a helper function that takes as a parameter the number of nodes to recover. This helper function gets called thrice (once each for recovering 0, 1, and 2). The logic is almost identical to TestLogReplicationSingleFollowerFailureThenRecovery, except that we could recover a different number of followers than exactly 1 follower.
6. TestLogReplicationLeaderFailureThenRecovery: Tests the complete cycle of leader election, committing some operations, then leader failure, new leader election, continued operations under the new leader, and successful recovery when the original leader rejoins as a follower. After the old leader gets restarted, we verify that all nodes end up with the same state by calling waitForCommitIndex and waitForLogReplication with timeouts set to 5 seconds. Finally, we submit one more operation and then call waitForCommitIndex and waitForLogReplication with timeouts set to 5 seconds to ensure everything can still move forward. Finally, we verify the nodes' states are as expected.
7. TestLogReplicationLeaderFailureWithUncommittedThenRecovery: Tests the scenario where a leader fails with uncommitted log entries. The logic for this test is extremely similar to TestLogReplicationLeaderFailureThenRecovery, except that to simulate the uncommitted log entries scenario, after we crash the leader, we overwrite its persisted state with extra operations.
8. TestLogReplicationAllNodesCrashAndRecover: Tests persistence even if all nodes crash. We first elect a leader and submit some operations, then crash all nodes. Then all nodes are restarted, and more operations are submitted. We check to make sure each node's final state also includes the state that it had before it crashed.

4-3. Log Compaction Tests:

For all tests in this section, the election timeout values by default range from ~2.5 to ~3.5 seconds, and heartbeats roughly get sent every ~1 second. Same as the Leader Election Tests, we also use the ServeBackend function to start Raft nodes and the corresponding gRPC servers. The log compaction threshold was set to 5 in all tests. For all tests, after electing a leader, we call **waitForStableLeader** with a timeout duration of 10 seconds.

We wrote **6 tests** to test our log compaction correctness:

1. TestLogCompactionFollowerRestartsButNotTooFarBehind: Tests log compaction when a follower goes down but isn't too far behind when it recovers. We set the compaction threshold to 5 entries, submit 7 operations (5 Sets and 1 Delete), wait for snapshots to be created and logs to be compacted on all nodes, then crash one follower. After submitting one more operation to the remaining nodes, we restart the crashed follower and verify that it can catch up to the one log it missed (we wait for 5 seconds). Finally, we verify that all nodes have the correct final state.

2. TestLogCompactionFollowerRestartsButAtLeastOneSnapshotBehind: Tests log compaction when a follower goes offline and falls behind by at least one snapshot period. We submit 7 initial operations to trigger compaction, crash a follower, then submit 6 more operations to ensure triggering another round of compaction. When the follower restarts, it's far enough behind that it needs snapshot transfer rather than just log replication to catch up. We wait 10 seconds (so that the follower can get all resources set up and catch up) before checking that all node states are as expected.
3. TestLogCompactionFollowerJoinsAndMultipleSnapshotsBehind: Tests recovery when a follower is multiple snapshots behind. We crash a follower early, then submit operations in 3 batches of 5 operations each, causing multiple rounds of compaction and snapshot creation on the remaining nodes. When the follower rejoins, it must recover from being several snapshots behind. We wait 10 seconds before checking that all node states are as expected. Then we submit one more operation, wait a few seconds, and check all the node states again to make sure all states are as expected.
4. TestLogCompactionFollowerLosesPersistentData: Tests recovery when a follower completely loses its persistent storage (logs and snapshots). We trigger compaction, then shut down a follower and delete all its log files. After submitting more operations, we restart the follower with no persistent state, forcing it to recover entirely through snapshot transfer from other nodes. We wait for 10 seconds for the recovered follower to catch up, then check all the node states again to make sure all states are as expected.
5. TestConcurrentCompactionAndInstallation: Tests concurrent snapshot installation with multiple followers recovering simultaneously. Using a 5-node cluster, we crash 2 followers, then start submitting 20 operations concurrently in a goroutine to trigger snapshot creation. After waiting for 1 second, we then restart both crashed followers concurrently using goroutines to test parallel snapshot installation (while the leader is likely compacting). We then submit 10 more operations and verify that all nodes achieve a consistent state (within 10 seconds). Then we submit one more final operation, wait a few seconds, and check all node states to ensure all are the same.
6. TestLogCompactionAfterLeaderFailure: Tests log compaction behavior during leadership changes. We submit operations to trigger compaction, then crash the leader and wait for a new leader to be elected (we wait 10 seconds for a new leader to be elected). Under the new leader, we submit more operations to trigger additional compaction, then restart the original leader and verify it can catch up as a follower (we wait for 10 seconds). Finally, we check all the node states again to make sure all states are as expected.

Closing Thoughts & Future Work

Over roughly ~3 weeks, we have had the opportunity to code Raft from scratch using Go and gRPC. From our comprehensive testing, we're fairly confident in our implementation. Our overall development experience has been positive. Throughout development, it was fairly straightforward to reason about coding decisions given the characteristics of Raft. For example,

the single-leader characteristic gave us a focus point that made handling various scenarios manageable. Leader election being separate from log replication means that it is easier to reason about errors and edge cases for each area separately, and also allows for incremental testing and development. We found the paper's explanation of the implementation and safety guarantees to be quite clear and convincing.

Our implementation is simple to focus on the overall process of Raft, but there remains many paths to improve such as trying to increase the performance, more features such as cluster membership changes, and client usability. If we had more time, we could have also ventured to implement Paxos and see our progress within a similar time frame. Maybe then we would have an even stronger opinion on the understandability of Raft. But even without such a comparison, we've gained an appreciation for Raft's understandability.

Acknowledgement

We'd like to thank Professor Snoeren and Zhiyuan Guo for guidance and support on this project, and for teaching the course 223B.

Citations

[1] https://raft.github.io/raft.pdf

[2] https://lamport.azurewebsites.net/pubs/paxos-simple.pdf