CS 7610 : Distributed Systems
Professor Nita-Rotaru

Raft : An Implementation in GoLang

Meredith Accum and Muhammad Ali
December 3, 2018

I. Introduction

Raft is a consensus algorithm designed for simplicity and ease of implementation [1]. In this project, we write a complete implementation of Raft that is able to provide consensus for a key-value store replicated across multiple servers. Our implementation is written in Go. Our choice for Go was because of the features it provides to support concurrent programming – such as goroutines, channels and easy to use mutexes.

II. Implementation of Proposed Plan

In our project proposal, we planned to implement leader elections and log replication, both of which we have successfully implemented.

The four phases of the project (excluding stretch goals) that we included in our weekly plan were:

1. Infrastructure: RPCs Calls, TCP dialing, struct for storing Raft state
2. Leader Election
3. Log Replication
4. Persistent State : saving the state machine in a file for crash recovery

We successfully implemented all four stages and tested our program under the assumption that no network partitions occur.

III. Partner Responsibilities

We collaborated via a github repository, with our shared code. Meredith coded the initial infrastructure for RPC calls and Ali coded the initial infrastructure for heartbeats and timeouts. From then on, we began meeting to do pair programming. The vast majority of code was written via pair programming, with only occasional bug fixes and refactoring of the code done separately. We alternated back-and-forth between who was typing at the computer each day we met.

Because sections of Raft have logic are intertwined with each other, it was important that both partners understand the entire logic – we also found pair programming to be particularly helpful for spotting bugs while writing code.

IV. System Architecture and Design Decisions

The program has two modes : a Raft mode and a Client mode. To run the program, start a cluster of nodes in the Raft mode, using the hosts specified in hostfile.txt, and then start up a client instance. See the README for more details on this. We've tested our Raft cluster with up to five instances and one client. The cluster assumes there are always an odd number of nodes for simplicity. The client will randomly contact a node in the cluster. If the node is the leader, the leader will handle the client's command. If the node is a follower, it will return the last known leader's address the client, who will store it. The client will time out if it cannot make contact with a leader.

To add a value to the key-value store, the client sends a (key, value) pair to a Raft leader to replicate across the cluster and the leader responds with an acknowledgement when it has *committed* the entry i.e. replicated it across a majority of nodes [1]. If the client is able to get in touch with the leader, but the leader is unable to replicate the entry on a majority of nodes (because some followers have crashed), the client will block indefinitely until it receives a definitive acknowledgement. This was a design decision we made in order to prioritize consistency for the client over top of availability. Under the assumption that a majority of Raft instances are always alive, the situation where the client blocks indefinitely will not occur. An alternative design decision would have been to have the client continue allowing user commands even though the client's operation was not yet saved, but we felt it was preferable to prioritize consistency over availability.

V. Challenges

One challenge we dealt with in our implementation was ensuring that if a leader died before a log entry was committed, the logs needed to stay consistent across the cluster. Consider the example in the diagram, where the leader received a key-value pair (6, foo) from the client and replicated the log entry on one follower. If this leader dies before committing the entry, there are two scenarios.

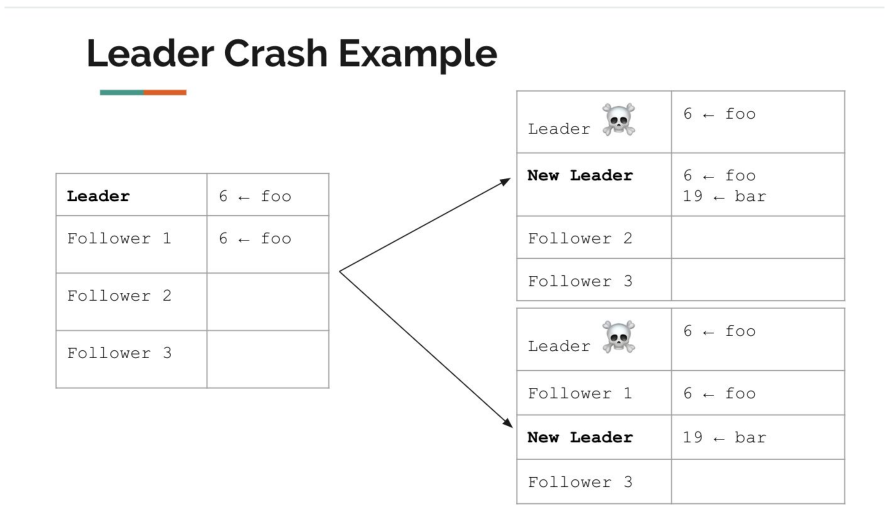

Figure 1: An example of the two possible recovery situations when the leader crashes after receiving an entry to replicate.

In the first scenario, the new elected leader could already have the (6, foo) key-value pair, in which case it should continue to try to replicate it. Initially, we misunderstood how the new leader should replicate this log entry. We thought this log entry would be replicated in the same manner as all other log entries, by replicating it on a majority of nodes. Instead, the old log entry with the old term number should actually be replicated via the Log Matching Property. The Raft paper describes this process in detail :

> "**Raft never commits log entries from previous terms by counting replicas.** Only log entries from the leader's current term are committed by counting replicas; once an entry from the current term has been committed this way, then all prior entries are committed **indirectly** because of the Log Matching Property" (page 9) [1]. (emphasis added)

Once we understood how the Log Matching Property was used to replicate these entries, we were able to ensure that the logs remained consistent in this case. In the second case in the diagram, the new leader does not have the (6,foo) entry in its log, so the (6,foo) entry must be overwritten on Follower 1's log entries. This case was easier to handle, because the (6,foo) entry can just be overwritten by the next (19, bar) entry.

VI. Final Thoughts

We learned that Raft is quite nuanced, despite being considered a simpler algorithm to understand than PAXOS. Particularly, the leader election was challenging for us to implement, although the log replication went much more smoothly once we started planning our code on paper before starting an implementation. We are content with our choice to implement the program in Go. The language had useful constructs for threading, timers, and RPC calls, which meant we had access to all the tools we needed to implement Raft. We pair programmed the algorithm, and without pair programming, the project would likely have taken us significantly longer, so this was a lesson well learned in effective programming practices.

References

[1] Ongaro, Diego, and John K. Ousterhout. "In Search of an Understandable Consensus Algorithm." *USENIX Annual Technical Conference*. 2014.