CS 378H Raft Project Report

To run:
> cd ./src
> go test

Most of the work is in src/raft.go

I. Background

With the developments of large scale data centers, many attempts have been made to achieve a strong center that never fails. Instead of building one super robust server, people find out it would be much cheaper and easier to build a reliable distributed system with commodity hardwires. However, it is well known that it is impossible to reach global consensus, or in other words, common knowledge, using unreliable communication channels such as networks. (Halpern) Implementing a distributed system in real life is prone to error due to its complexity and it is questionable whether the implementation faithfully followed the design of the protocol.

Raft is a distributed protocol that is easier to understand and implement among others and suitable to build practical systems. (Ongaro) It has one leader that takes requests from clients and issue tasks to the leader's followers. The leader sends out periodical heartbeat messages; once the follower has not received any heartbeat message for a certain amount of time, the follower will aspire to be the new leader and ask for majority votes. The wait time for each follower to start an election is randomized to prevent that all followers ask for a vote at the same time. After the server receives the majority vote, under Raft's assumption that more than half of the system is working, it becomes the new leader and propagates its log to the follower. The election is set up in a way that only one of the servers that have the most up-to-date log among surviving servers could become the leader. Raft uses file log as ground truth and coordinate its actions.

II. Code Structure

Each server has three states, follower, candidate and leader. Each follower will start a new election becoming candidate if no heart beat is heard after 250-450 milliseconds. Each candidate will ask for majority votes and start a new election cycle if the old one is not completed within 250-450 milliseconds. The leader will become a follower if it detects a server that has more up-to-date log, usually caused by previous network partition. The server state machine is best pictured in Figure 1.

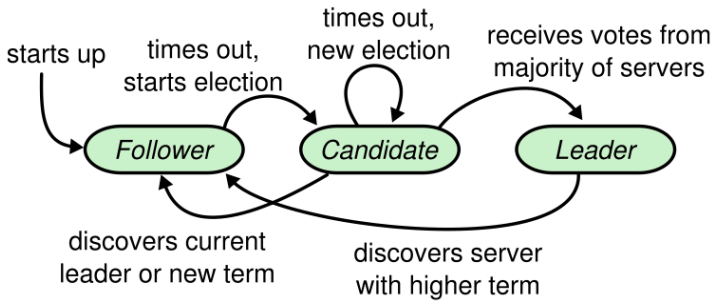

Figure 1, from Ongaro's paper

Each server has a separate go routine that reports the new log to test framework.

III. Implementation Hacks

Initially, the heartbeat messages were sent out in a go-routine that separates from the go-routine where the leader broadcasts new tasks to followers. However, for some reason, it keeps failing the majority of test cases. The heartbeat code is so concise that I do not think it is due to a bug in the implementation. After merging the heartbeat message with broadcasting task messages, the problem solves itself. The latter approach reduces half of the original network traffic, so I think it is the reason that solves the bug.

No go routines are intentionally killed as I assume go routine manager will take care of it after the parent thread quits.

IV. Major Limitation

The code is not concise enough that I have confidence that it does not have any issue. Instead of churning out test cases as best as we can, it'd be much elegant to have a proof of correctness.

Logging could be improved because the debug process mostly centers speculation and adding fmt.printf.

The current network model adopts a uniform delay distribution, which is different from modern data center network pattern. In reality, network delay is normal distribution with a heavy tail and we sometimes observe a burst of packet arrival due to possible router congestion. So a better network model will test the system in a more realistic way.

V. Reflection

It takes a surprisingly long time to debug the code, even though the core component is within 1000 lines of code. The most unforgettable bug I had was using *i* instead of *j* in a nested loop. They become much more similar to each other in brackets. The error could easily be caught at an early stage if unit testing was used. It is a real-life lesson that unit testing is my good friend.

VI. Citations

Halpern, Knowledge and Common Knowledge in a Distributed Environment, https://www.cs.utexas.edu/users/lorenzo/corsi/cs380d/papers/p549-halpern.pdf
Ongaro, In Search of an Understandable Consensus Algorithm, https://pdos.csail.mit.edu/6.824/papers/raft-extended.pdf