COS418 Assignment 3: Raft Leader Election

Introduction

This is the first in a series of assignments in which you'll build a fault-tolerant key/value storage system. You'll start in this assignment by implementing the leader election features of Raft, a replicated state machine protocol. In Assignment 3 you will complete Raft's log consensus agreement features. You will implement Raft as a Go object with associated methods, available to be used as a module in a larger service. Once you have completed Raft, the course assignments will conclude with such a service: a key/value service built on top of Raft.

Raft Overview

The Raft protocol is used to manage replica servers for services that must continue operation in the face of failure (e.g. server crashes, broken or flaky networks). The challenge is that, in the face of these failures, the replicas won't always hold identical data. The Raft protocol helps sort out what the correct data is.

Raft's basic approach for this is to implement a replicated state machine. Raft organizes client requests into a sequence, called the log, and ensures that all the replicas agree on the the contents of the log. Each replica executes the client requests in the log in the order they appear in the log, applying those requests to the service's state. Since all the live replicas see the same log contents, they all execute the same requests in the same order, and thus continue to have identical service state. If a server fails but later recovers, Raft takes care of bringing its log up to date. Raft will continue to operate as long as at least a majority of the servers are alive and can talk to each other. If there is no such majority, Raft will make no progress, but will pick up where it left off as soon as a majority is alive again.

You should consult the extended Raft paper and the Raft lecture notes. You may also find this illustrated Raft guide useful to get a sense of the high-level workings of Raft. For a wider perspective, have a look at Paxos, Chubby, Paxos Made Live, Spanner, Zookeeper, Harp, Viewstamped Replication, and Bolosky et al.

Software

For this assignment, we will focus primarily on the code and tests for the Raft implementation in `src/raft` and the simple RPC-like system in `src/labrpc`. It is worth your while to read and digest the code in these packages.

Before you have implemented anything, your raft tests will fail, but this behavior is a sign that you have everything properly configured and are ready to begin:

```
$ cd cos418 # or wherever you unpacked your tarball
$ export GOPATH="$PWD"
$ cd "$GOPATH/src/raft"
$ go test -run Election
Test: initial election ...
--- FAIL: TestInitialElection (5.00s)
config.go:286: expected one leader, got none
Test: election after network failure ...
--- FAIL: TestReElection (5.00s)
config.go:286: expected one leader, got none
FAIL
exit status 1
```

You should implement Raft by adding code to `raft/raft.go` (only). In that file you'll find a bit of skeleton code, plus some examples of how to send and receive RPCs, and examples of how to save and restore persistent state.

Your Task: Leader Election

You should start by reading the code to determine which functions are responsible for conducting Raft leader election, if you haven't already.

The natural first task is to fill in the `RequestVoteArgs` and `RequestVoteReply` structs, and modify `Make()` to create a background goroutine that starts an election (by sending out `RequestVote` RPCs) when it hasn't heard from another peer for a while. For election to work, you will also need to implement the `RequestVote()` RPC handler so that servers will vote for one another.

To implement heartbeats, you will need to define an `AppendEntries` RPC struct (though you will not need any real payload yet), and have the leader send them out periodically. You will also have to write an `AppendEntries` RPC handler method that resets the election timeout so that other servers don't step forward as leaders when one has already been elected.

Make sure the timers in different Raft peers are not synchronized. In particular, make sure the election timeouts don't always fire at the same time, or else all peers will vote for themselves and no one will become leader.

Your Raft implementation must support the following interface, which the tester and (eventually) your key/value server will use. You'll find more details in comments in `raft.go`.

```
// create a new Raft server instance:
rf := Make(peers, me, persister, applyCh)

// start agreement on a new log entry:
rf.Start(command interface{}) (index, term, isleader)

// ask a Raft for its current term, and whether it thinks it is leader
rf.GetState() (term, isLeader)

// each time a new entry is committed to the log, each Raft peer
// should send an ApplyMsg to the service (or tester).
type ApplyMsg
```

A service calls `Make(peers,me,…)` to create a Raft peer. The peers argument is an array of established RPC connections, one to each Raft peer (including this one). The `me` argument is the index of this peer in the peers array. `Start(command)` asks Raft to start the processing to append the command to the replicated log. `Start()` should return immediately, without waiting for for this process to complete. The service expects your implementation to send an `ApplyMsg` for each new committed log entry to the `applyCh` argument to `Make()`.

Your Raft peers should exchange RPCs using the labrpc Go package that we provide to you. It is modeled after Go's rpc library, but internally uses Go channels rather than sockets. `raft.go` contains some example code that sends an RPC (`sendRequestVote()`) and that handles an incoming RPC (`RequestVote()`).

Implementing leader election and heartbeats (empty `AppendEntries` calls) should be sufficient for a single leader to be elected and -- in the absence of failures -- stay the leader, as well as redetermine leadership after failures. Once you have this working, you should be able to pass the two Election "go test" tests:

```
$ go test -run Election
Test: initial election ...
  ... Passed
Test: election after network failure ...
  ... Passed
PASS
  ok  raft7.008s
```

Resources and Advice

- Start early. Although the amount of code to implement isn't large, getting it to work correctly will be very challenging. Both the algorithm and the code is tricky and there are many corner cases to consider. When one of the tests fails, it may take a bit of puzzling to understand in what scenario your solution isn't correct, and how to fix your solution.
- Read and understand the extended Raft paper and the Raft lecture notes before you start. Your implementation should follow the paper's description closely, since that's what the tests expect. Figure 2 may be useful as a pseudocode reference.
- Add any state you need to keep to the `Raft` struct in `raft.go`. Figure 2 in the paper may provide a good guideline.

Submission

Submit your code to the COS418 Assignment 3 Dropbox by 11:59pm on 11/21. You may submit multiple times, only the one in the Dropbox at the time of grading will be recorded.

Before submitting, please run the full tests given above for both parts one final time. You will receive full credit for the leader election component if your software passes the Election tests (as run by the `go test` commands above) on the CS servers.

The final portion of your credit is determined by code quality tests, using the standard tools `gofmt` and `go vet`. You will receive full credit for this portion if all files submitted conform to the style standards set by `gofmt` and the report from `go vet` is clean for your raft package (that is, produces no errors). If your code does not pass the `gofmt` test, you should reformat your code using the tool. You can also use the Go Checkstyle tool for advice to improve your code's style, if applicable. Additionally, though not part of the graded checks, it would also be advisable to produce code that complies with Golint where possible.

Acknowledgements

This assignment is adapted from MIT's 6.824 course. Thanks to Frans Kaashoek, Robert Morris, and Nickolai Zeldovich for their support.

Last updated: 2017-10-30 16:21:53 -0400