6.824 - Spring 2016

6.824 Lab 2: Raft

Due: Fri Feb 26, 11:59pm

Introduction

This is the first in a series of labs in which you'll build a fault-tolerant key/value storage system. You'll start in this lab by implementing Raft, a replicated state machine protocol. In the next lab lab you'll build a key/value service on top of Raft. Then you will “shard” your service for higher performance, and finally implement transactional operations across shards.

A replicated service (e.g., key/value database) uses Raft to help manage its replica servers. The point of having replicas is so that the service can continue operating even if some of the replicas experience failures (crashes or a broken or flaky network). The challenge is that, due to these failures, the replicas won't always hold identical data; Raft helps the service sort out what the correct data is.

Raft's basic approach 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.

In this lab you'll implement Raft in the form of Go object type with associated methods, meant to be used as a module in a larger service. A set of Raft instances talk to each other with RPC to maintain replicated logs. Your Raft interface will support an indefinite sequence of numbered commands, also called log entries. The entries are numbered with *index numbers*. The log entry with a given index will eventually be committed. At that point, your Raft should send the log entry to the larger service for it to execute.

> **Note:** Only RPC may be used for interaction between different Raft instances. For example, different instances of your Raft implementation are not allowed to share Go variables. Your implementation should not use files at all.

In this lab you'll implement most of the Raft design described in the extended paper, including saving persistent state and reading it after a node fails and then restarts. You will not implement cluster membership changes (Section 6) or log compaction / snapshotting (Section 7).

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.

- **Hint:** 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.
- **Hint:** 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.

Collaboration Policy

You must write all the code you hand in for 6.824, except for code that we give you as part of the assignment. You are not allowed to look at anyone else's solution, you are not allowed to look at code from previous years, and you are not allowed to look at other Raft implementations. You may discuss the assignments with other students, but you may not look at or copy each others' code. Please do not publish your code or make it available to future 6.824 students -- for example, please do not make your code visible on GitHub (instead, create a private repository on MIT's GitHub deployment).

Getting Started

Do a `git pull` to get the latest lab software. We supply you with skeleton code and tests in `src/raft`, and a simple RPC-like system in `src/labrpc`.

To get up and running, execute the following commands:

```
$ setup ggo_v1.5
$ cd ~/6.824
$ git pull
...
$ cd src/raft
$ GOPATH=~/6.824
$ export GOPATH
$ go test
Test: initial election ...
--- FAIL: TestInitialElection (5.03s)
        config.go:270: expected one leader, got 0
Test: election after network failure ...
--- FAIL: TestReElection (5.03s)
        config.go:270: expected one leader, got 0
...
$
```

When you're done, your implementation should pass all the tests in the `src/raft` directory:

```
$ go test
Test: initial election ...
  ... Passed
Test: election after network failure ...
  ... Passed
...
PASS
ok      raft    162.413s
```

Your Job

You should implement Raft by adding code to `raft/raft.go`. 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 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()`).

TASK

Implement leader election and heartbeats (empty `AppendEntries` calls). This should be sufficient for a single leader to be elected, and to stay the leader, in the absence of failures. Once you have this working, you should be able to pass the first two "go test" tests.

- **Hint:** Add any state you need to keep to the `Raft` struct in `raft.go`. Figure 2 in the paper may provide a good guideline. You'll also need to define a struct to hold information about each log entry. Remember that the field names any structures you will be sending over RPC must start with capital letters, as must the field names in any structure passed inside an RPC.
- **Hint:** You should start by implementing Raft leader election. 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.
- **Hint:** To implement heartbeats, you will need to define an `AppendEntries` RPC

structs (though you may not need all the arguments 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.
- **Hint:** 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.

While being able to elect a leader is useful, we want to use Raft to keep a consistent, replicated log of operations. To do so, we need to have the servers accept client operations through `Start()`, and insert them into the log. In Raft, only the leader is allowed to append to the log, and should disseminate new entries to other servers by including them in its outgoing `AppendEntries` RPCs.

TASK

Implement the leader and follower code to append new log entries. This will involve implementing `Start()`, completing the `AppendEntries` RPC structs, sending them, and fleshing out the `AppendEntry` RPC handler. Your goal should first be to pass the `TestBasicAgree()` test (in `test_test.go`). Once you have that working, you should try to get all the tests before the "basic persistence" test to pass.

- **Hint:** A significant portion of this lab will consist of making your implementation robust against various kinds of failures. You will need to implement the election restriction (section 5.4.1 in the paper). The next set of tests have to do with various failure cases, in which some servers don't receive some RPCs and some servers occasionally crash and restart.
- **Hint:** While the Raft leader is the only server that causes entries to be appended to the log, all the servers need to independently give newly committed entries to their local service replica (via their own `applyCh`). Because of this, you should try to keep these two activities as separate as possible.
- **Hint:** Figure out the minimum number of messages Raft should use when reaching agreement in non-failure cases and make your implementation use that minimum.

A Raft-based server must be able to pick up where it left off, and continue if the computer it's on reboots. This requires that Raft keep persistent state that survives a reboot (the paper's Figure 2 mentions which state should be persistent).

A “real” implementation would do this by writing Raft's persistent state to disk each time it changes, and reading the latest saved state from disk when restarting after a reboot. Your implementation won't use the disk; instead, it will save and restore persistent state from a `Persister` object (see `persister.go`). Whoever calls `Make()` supplies a `Persister` that initially holds Raft's most recently persisted state (if any). Raft should initialize its state from that `Persister`, and should use it to save its persistent state each time the state changes. You can use the `ReadRaftState()` and `SaveRaftState()` methods for this respectively.

TASK

Implement persistence by first adding code to serialize any state that needs persisting in `persist()`, and to unserialize that same state in `readPersist()`. You now need to determine at what points in the Raft protocol your servers are required to persist their state, and insert calls to `persist()` in those places. Once this code is complete, you should pass the remaining tests. You may want to first try and pass the "basic persistence" test (`go test -run`

`'TestPersist1$'`), and then tackle the remaining ones.

Note: You will need to encode the state as an array of bytes in order to pass it to the `Persister`; `raft.go` contains some example code for this in `persist()` and `readPersist()`.

Note: In order to avoid running out of memory, Raft must periodically discard old log entries, but you **do not** have to worry about garbage collecting the log in this lab. You will implement that in the next lab by using snapshotting (Section 7 in the paper).

- **Hint:** Similarly to how the RPC system only sends structure field names that begin with upper-case letters, and silently ignores fields whose names start with lower-case letters, the GOB encoder you'll use to save persistent state only saves fields whose names start with upper case letters. This is a common source of mysterious bugs, since Go doesn't warn you.
- **Hint:** In order to pass some of the challenging tests towards the end, such as those marked "unreliable", you will need to implement the optimization to allow a follower to back up the leader's nextIndex by more than one entry at a time. See the description in the extended Raft paper starting at the bottom of page 7 and top of page 8 (marked by a gray line).

Handin procedure

Important:

Before submitting, please run *all* the tests one final time. **You** are responsible for making sure your code works. Keep in mind that the more obscure corner cases may not appear on every run, so it's a good idea to run the tests multiple times.

```
$ go test
```

Submit your code via the class's submission website, located at https://6824.scripts.mit.edu:444/submit/handin.py/.

You may use your MIT Certificate or request an API key via email to log in for the first time. Your API key (`XXX`) is displayed once you logged in, which can be used to upload the lab from the console as follows.

```
$ cd "$GOPATH"
$ echo "XXX" > api.key
$ make lab2
```

Important:

Check the submission website to make sure you submitted a working lab!

Note: You may submit multiple times. We will use the timestamp of your **last** submission for the purpose of calculating late days. Your grade is determined by the score your solution **reliably** achieves when we run the tester on our test machines.

Please post questions on Piazza.