CS443 - Fall 2018

Lab 1, 2: Raft

Lab 1 (Leader Election) Due: Nov 11 at 11:59pm

Lab 2 (Append Entries) Due: Nov 22 at 11:59pm

Introduction

These labs are from MIT cource 6.824: Distributed Systems.

This is the first in a series of labs in which you'll build a fault-tolerant key/value storage system. In the first lab you will build the leader election mechanism in Raft, a replicated state machine protocol. In the second lab you will implement the entry append mechanism. In the final lab you will build a key/value service on top of Raft.

For those who like challenges, we have included a challenge section at the end of the second lab, in which you will add persistence into Raft.

A replicated service achieves fault tolerance by storing complete copies of its state (i.e., data) on multiple replica servers. Replication allows the service to continue operating even if some of its servers experience failures (crashes or a broken or flaky network). The challenge is that failures may cause the replicas to hold differing copies of the data.

Raft manages a service's state replicas, and in particular it helps the service sort out what the correct state is after failures. Raft implements a replicated state machine. It organizes client requests into a sequence, called the log, and ensures that all the replicas agree on 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 replica's local copy of 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 can communicate again.

In this lab you'll implement Raft as a 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:** Your Raft instances are only allowed to interact using RPC. For example, different Raft instances are not allowed to share Go variables. Your code should not use files at all.

You should consult the extended Raft paper and the Raft lecture notes. You may find it useful to look at this illustration of the Raft protocol, a guide to Raft implementation, and advice about locking and structure for concurrency. For a wider perspective, have a look at Paxos, Chubby, Paxos Made Live, Spanner, Zookeeper, Harp, Viewstamped Replication, and Bolosky et al.

In this lab you'll implement most of the Raft design described in the extended paper, but we leave the following as bonus challenges only for those who fell up to them: 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).

- **Hint:** Start early. Although the amount of code isn't large, getting it to work correctly will be challenging.
- **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, particularly Figure 2, since that's what the tests expect.

This lab does not involve a lot of code, but concurrency makes it challenging to debug; start each part early.

Collaboration Policy

COPYING WILL BE GRADED "F"

You must write all the code you hand in for CS443, 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 anyone else's code, or allow anyone else to look at your code.

Please do not publish your code or make it available to current or future CS443 students. `github.com` repositories are public by default, so please don't put your code there unless you make the repository private.

Getting Started

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:

```
$ git clone https://github.com/ANLAB-KAIST/CS443-RaftKV.git CS443
...
$ cd src/raft
$ GOPATH=~/CS443
$ export GOPATH
$ go test
Test (Lab1): initial election ...
--- FAIL: TestInitialElectionLab1 (5.04s)
        config.go:305: expected one leader, got none
Test (Lab1): election after network failure ...
--- FAIL: TestReElectionLab1 (5.03s)
        config.go:305: expected one leader, got none
...
$
```

The code

Implement Raft by adding code to `raft/raft.go`. In that file you'll find a bit of skeleton code, plus examples of how to send and receive RPCs.

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 network identifiers of the Raft peers (including this one), for use with labrpc RPC. 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 the log appends to complete. The service expects your implementation to send an `ApplyMsg` for each newly 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()`). The reason you must use `labrpc` instead of Go's RPC package is that the tester tells `labrpc` to delay RPCs, re-order them, and delete them to simulate challenging network conditions under which your code should work correctly. Don't rely on modifications to `labrpc` because we will test your code with the `labrpc` as handed out.

Your first implementation may not be clean enough that you can easily reason about its correctness. Give yourself enough time to rewrite your implementation so that you can easily reason about its correctness. Subsequent labs will build on this lab, so it is important to do a good job on your implementation.

Lab 1

TASK

Implement leader election and heartbeats (`AppendEntries` RPCs with no log entries). The goal for Part Lab1 is for a single leader to be elected, for the leader to remain the leader if there are no failures, and for a new leader to take over if the old leader fails or if packets to/from the old leader are lost. Run `go test -run Lab1` to test your Lab1 code.

- **Hint:** Add any state you need to the `Raft` struct in `raft.go`. You'll also need to define a struct to hold information about each log entry. Your code should follow Figure 2 in the paper as closely as possible.
- **Hint:** Fill in the `RequestVoteArgs` and `RequestVoteReply` structs. Modify `Make()` to create a background goroutine that will kick off leader election periodically by sending out `RequestVote` RPCs when it hasn't heard from another peer for a while. This way a peer will learn who is the leader, if there is already a leader, or become the leader itself. Implement the `RequestVote()` RPC handler so that servers will vote for one another.
- **Hint:** To implement heartbeats, define an `AppendEntries` RPC struct (though you may not need all the arguments yet), and have the leader send them out periodically. 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 election timeouts in different peers don't always fire at the same time, or else all peers will vote only for themselves and no one will become the leader.
- **Hint:** The tester requires that the leader send heartbeat RPCs no more than ten times per second.
- **Hint:** The tester requires your Raft to elect a new leader within five seconds of the failure of the old leader (if a majority of peers can still communicate). Remember, however, that leader election may require multiple rounds in case of a split vote (which can happen if packets are lost or if candidates unluckily choose the same random backoff times). You must pick election timeouts (and thus heartbeat intervals) that are short enough that it's very likely that an election will complete in less than five seconds even if it requires multiple rounds.
- **Hint:** The paper's Section 5.2 mentions election timeouts in the range of 150 to 300 milliseconds. Such a range only makes sense if the leader sends heartbeats considerably more often than once per 150 milliseconds. Because the tester limits you to 10 heartbeats per second, you will have to use an election timeout larger than the paper's 150 to 300 milliseconds, but not too large, because then you may fail to elect a leader within five seconds.
- **Hint:** You may find Go's rand useful.

- **Hint:** You'll need to write code that takes actions periodically or after delays in time. The easiest way to do this is to create a goroutine with a loop that calls time.Sleep(). The hard way is to use Go's `time.Timer` or `time.Ticker`, which are difficult to use correctly.
- **Hint:** If you are puzzled about locking, you may find this advice helpful.
- **Hint:** If your code has trouble passing the tests, read the paper's Figure 2 again; the full logic for leader election is spread over multiple parts of the figure.
- **Hint:** A good way to debug your code is to insert print statements when a peer sends or receives a message, and collect the output in a file with `go test -run Lab1 > out`. Then, by studying the trace of messages in the `out` file, you can identify where your implementation deviates from the desired protocol. You might find `DPrintf` in `util.go` useful to turn printing on and off as you debug different problems.
- **Hint:** Go RPC sends only struct fields whose names start with capital letters. Sub-structures must also have capitalized field names (e.g. fields of log records in an array). The `labgob` package will warn you about this; don't ignore the warnings.
- **Hint:** You should check your code with `go test -race`, and fix any races it reports.

Be sure you pass the Lab1 tests before submitting Part Lab1, so that you see something like this:

```
$ go test -run Lab1
Test (Lab1): initial election ...
  ... Passed --   2.5  3   30    0
Test (Lab1): election after network failure ...
  ... Passed --   4.5  3   78    0
PASS
ok      raft    7.016s
$
```

Each "Passed" line contains four numbers; these are the time that the test took in seconds, the number of Raft peers (usually 3 or 5), the number of RPCs sent during the test, and the number of log entries that Raft reports were committed. Your numbers will differ from those shown here. You can ignore the numbers if you like, but they may help you sanity-check the number of RPCs that your implementation sends. For all of labs 2, 3, and 4, the grading script will fail your solution if it takes more than 600 seconds for all of the tests (`go test`), or if any individual test takes more than 120 seconds.

Lab 2 and Challenge 1 will test leader election in more challenging settings and may expose bugs in your leader election code which the Lab1 tests miss.

Important:

Before submitting, please run the 2A tests one final time. Some bugs may not appear on every run, so run the tests multiple times.

Lab 2

We want Raft to keep a consistent, replicated log of operations. A call to `Start()` at the leader starts the process of adding a new operation to the log; the leader sends the new operation to the other servers in `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, fleshing out the `AppendEntry` RPC handler, and advancing the `commitIndex` at the leader. Your first goal should be to pass the `TestBasicAgreeLab2()` test (in `test_test.go`). Once you have that working, you should get all the Lab2 tests to pass (`go test -run Lab2`).

- **Hint:** You will need to implement the election restriction (section 5.4.1 in the paper).
- **Hint:** One way to fail the early Lab Lab2 tests is to hold un-needed elections, that is, an election even though the current leader is alive and can talk to all peers. This can prevent agreement in situations where the tester believes agreement is possible. Bugs in election timer management, or not sending out heartbeats immediately after winning an election, can cause un-needed elections.
- **Hint:** You may need to write code that waits for certain events to occur. Do not write loops that execute continuously without pausing, since that will slow your implementation enough that it fails tests. You can wait efficiently with Go's channels, or Go's condition variables, or (if all else fails) by inserting a `time.Sleep(10 * time.Millisecond)` in each loop iteration.
- **Hint:** Give yourself time to rewrite your implementation in light of lessons learned about structuring concurrent code. In later labs you'll thank yourself for having Raft code that's as clear and clean as possible. For ideas, you can re-visit our structure, locking and guide, pages.

The tests for upcoming labs may fail your code if it runs too slowly. You can check how much real time and CPU time your solution uses with the time command. Here's some typical output for Lab 2:

```
$ time go test -run Lab2
Test (Lab2): basic agreement ...
  ... Passed --   0.5  5   28    3
Test (Lab2): agreement despite follower disconnection ...
  ... Passed --   3.9  3   69    7
Test (Lab2): no agreement if too many followers disconnect ...
  ... Passed --   3.5  5  144    4
Test (Lab2): concurrent Start()s ...
  ... Passed --   0.7  3   12    6
Test (Lab2): rejoin of partitioned leader ...
  ... Passed --   4.3  3  106    4
Test (Lab2): leader backs up quickly over incorrect follower logs ...
  ... Passed --  23.0  5 1302  102
Test (Lab2): RPC counts aren't too high ...
  ... Passed --   2.2  3   30   12
PASS
ok      raft    38.029s
```

```
real    0m38.511s
user    0m1.460s
sys     0m0.901s
$
```

The "ok raft 38.029s" means that Go measured the time taken for the Lab2 tests to be 38.029 seconds of real (wall-clock) time. The "user 0m1.460s" means that the code consumed 1.460 seconds of CPU time, or time spent actually executing instructions (rather than waiting or sleeping). If your solution uses much more than a minute of real time for the Lab2 tests, or much more than 5 seconds of CPU time, you may run into trouble later on. Look for time spent sleeping or waiting for RPC timeouts, loops that run without sleeping or waiting for conditions or channel messages, or large numbers of RPCs sent.

Be sure you pass the Lab1 and Lab2 tests before submitting Part Lab2.

Challenge 1

> **Important:**
>
> Challenges are not mandatory. However, if you finish challenges, you will get bonus points

If a Raft-based server reboots it should resume service where it left off. This requires that Raft keep persistent state that survives a reboot. The paper's Figure 2 mentions which state should be persistent, and `raft.go` contains examples of how to save and restore persistent state.

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 `Raft.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. Use the `Persister`'s `ReadRaftState()` and `SaveRaftState()` methods.

TASK

Complete the functions `persist()` and `readPersist()` in `raft.go` by adding code to save and restore persistent state. You will need to encode (or "serialize") the state as an array of bytes in order to pass it to the `Persister`. Use the `labgob` encoder we provide to do this; see the comments in `persist()` and `readPersist()`. `labgob` is derived from Go's `gob` encoder; the only difference is that `labgob` prints error messages if you try to encode structures with lower-case field names.

TASK

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. There is already a call to `readPersist()` in `Raft.Make()`. Once you've done this, you should pass the remaining tests. You may want to first try to pass the "basic persistence" test (`go test -run 'TestPersist1Challenge1'`), and then tackle the remaining ones (`go test -run Challenge1`).

Note: In order to avoid running out of memory, Raft must periodically discard old log entries, but you **do not** have to worry about this until the next lab.

- **Hint:** Many of the Challenge1 tests involve servers failing and the network losing RPC requests or replies.
- **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). The paper is vague about the details; you will need to fill in the gaps, perhaps with the help of the CS443 Raft lectures.
- **Hint:** A reasonable amount of time to consume for the full set of tests(Lab1+Lab2+Challenge1) is 4 minutes of real time and one minute of CPU time.

Your code should pass all the Challenge1 tests (as shown below), as well as the Lab1 and Lab2 tests.

```
$ go test -run Challenge1
Test (Challenge1): basic persistence ...
  ... Passed --   3.4  3   60    6
Test (Challenge1): more persistence ...
  ... Passed --  17.0  5  705   16
Test (Challenge1): partitioned leader and one follower crash, leader restarts ...
  ... Passed --   1.4  3   27    4
Test (Challenge1): Figure 8 ...
  ... Passed --  33.2  5  852   53
Test (Challenge1): unreliable agreement ...
  ... Passed --   2.4  5  207  246
Test (Challenge1): Figure 8 (unreliable) ...
  ... Passed --  35.3  5 1838  216
Test (Challenge1): churn ...
  ... Passed --  16.2  5 5138 2260
Test (Challenge1): unreliable churn ...
  ... Passed --  16.2  5 1254  603
PASS
ok      raft    124.999s
```