[6.824](#) - Spring 2020

6.824 Lab 2: Raft

Part 2A Due: Feb 21 23:59

Part 2B Due: Feb 28 23:59

Part 2C Due: Mar 6 23:59

Introduction

This is the first in a series of labs in which you'll build a fault-tolerant key/value storage system. In this lab you'll implement Raft, a replicated state machine protocol. In the next lab you'll build a key/value service on top of Raft. Then you will "shard" your service over multiple replicated state machines for higher performance.

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 organizes client requests into a sequence, called the log, and ensures that all the replica servers see the same log. Each replica executes client requests in log order, applying them to its 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.

You should follow the design in the [extended Raft paper](#), with particular attention to Figure 2. You'll implement most of what's in the paper, including saving persistent state and reading it after a node fails and then restarts. You will not implement cluster membership changes (Section 6). You'll implement log compaction / snapshotting (Section 7) in a later lab.

You may find this [guide](#) useful, as well as this 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.](#)

This lab is due in three parts. You must submit each part on the corresponding due date.

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 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 6.824 students. `github.com` repositories are public by default, so please don't put your code there unless you make the repository private. You may find it convenient to use [MIT's GitHub](), but be sure to create a private repository.

Getting Started

If you have done Lab 1, you already have a copy of the lab source code. If not, you can find directions for obtaining the source via git in the [Lab 1 instructions]().

We supply you with skeleton code `src/raft/raft.go`. We also supply a set of tests, which you should use to drive your implementation efforts, and which we'll use to grade your submitted lab. The tests are in `src/raft/test_test.go`.

To get up and running, execute the following commands. Don't forget the `git pull` to get the latest software.

```
$ cd ~/6.824
$ git pull
...
$ cd src/raft
$ go test
Test (2A): initial election ...
--- FAIL: TestInitialElection2A (5.04s)
        config.go:326: expected one leader, got none
Test (2A): election after network failure ...
--- FAIL: TestReElection2A (5.03s)
        config.go:326: expected one leader, got none
...
$
```

The code

Implement Raft by adding code to `raft/raft.go`. In that file you'll find 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 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` channel argument to `Make()`.

`raft.go` contains example code that sends an RPC (`sendRequestVote()`) and that handles an incoming RPC (`RequestVote()`). Your Raft peers should exchange RPCs using the labrpc Go package (source in `src/labrpc`). The tester can tell `labrpc` to delay RPCs, re-order them, and discard them to simulate various network failures. While you can temporarily modify `labrpc`, make sure your Raft works with the original `labrpc`, since that's what we'll use to test and grade your lab. Your Raft instances must interact only with RPC; for example, they are not allowed to communicate using shared Go variables or files.

Subsequent labs build on this lab, so it is important to give yourself enough time to write solid code.

Part 2A

> Implement Raft leader election and heartbeats (`AppendEntries` RPCs with no   **TASK**
> log entries). The goal for Part 2A 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 2A` to test
> your 2A code.

- **Hint:** You can't easily run your Raft implementation directly; instead you should run it by way of the tester, i.e. `go test -run 2A`.
- **Hint:** Follow the paper's Figure 2. At this point you care about sending and receiving RequestVote RPCs, the Rules for Servers that relate to elections, and the State related to leader election,
- **Hint:** Add the Figure 2 state for leader election to the `Raft` struct in `raft.go`. You'll also need to define a struct to hold information about each log entry.
- **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()](#). Don't use Go's `time.Timer` or `time.Ticker`, which are difficult to use correctly.
- **Hint:** Read this advice about [locking](#) and [structure](#).
- **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:** Don't forget to implement `GetState()`.
- **Hint:** The tester calls your Raft's `rf.Kill()` when it is permanently shutting down an instance. You can check whether `Kill()` has been called using `rf.killed()`. You may want to do this in all loops, to avoid having dead Raft instances print confusing messages.
- **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 2A > 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:** Check your code with `go test -race`, and fix any races it reports.

Be sure you pass the 2A tests before submitting Part 2A, so that you see something like this:

```
$ go test -run 2A
Test (2A): initial election ...
  ... Passed --   4.0  3   32    9170    0
Test (2A): election after network failure ...
  ... Passed --   6.1  3   70   13895    0
PASS
ok      raft    10.187s
$
```

Each "Passed" line contains five 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, the total number of bytes in the RPC messages, 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.

Handin procedure for lab 2A

First, please run the 2A tests one last time. Then, run `make lab2a` to upload your code to the [submission site](#).

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 are logged in, and can be used to upload the lab from the console as follows.

```
$ cd ~/6.824
$ echo "XXX" > api.key
$ make lab2a
```

Check the submission website to make sure it sees your submission.

You may submit multiple times. We will use your **last** submission to calculate late days. Your grade is determined by the score your solution **reliably** achieves when we run the tester.

Part 2B

> **TASK**
>
> Implement the leader and follower code to append new log entries, so that the `go test -run 2B` tests pass.

- **Hint:** Run `git pull` to get the latest lab software.
- **Hint:** Your first goal should be to pass `TestBasicAgree2B()`. Start by implementing `Start()`, then write the code to send and receive new log entries via `AppendEntries` RPCs, following Figure 2.
- **Hint:** You will need to implement the election restriction (section 5.4.1 in the paper).
- **Hint:** One way to fail to reach agreement in the early Lab 2B tests is to hold repeated elections even though the leader is alive. Look for bugs in election timer management, or not sending out heartbeats immediately after winning an election.
- **Hint:** Your code may have loops that repeatedly check for certain events. Don't have these loops execute continuously without pausing, since that will slow your implementation enough that it fails tests. Use Go's [condition variables](#), or insert a `time.Sleep(10 * time.Millisecond)` in each loop iteration.
- **Hint:** Do yourself a favor for future labs and write (or re-write) code that's clean and clear. 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 typical output:

```
$ time go test -run 2B
Test (2B): basic agreement ...
  ... Passed --   1.6  3    18    5158     3
Test (2B): RPC byte count ...
  ... Passed --   3.3  3    50  115122    11
Test (2B): agreement despite follower disconnection ...
  ... Passed --   6.3  3    64   17489     7
Test (2B): no agreement if too many followers disconnect ...
  ... Passed --   4.9  5   116   27838     3
Test (2B): concurrent Start()s ...
  ... Passed --   2.1  3    16    4648     6
Test (2B): rejoin of partitioned leader ...
  ... Passed --   8.1  3   111   26996     4
Test (2B): leader backs up quickly over incorrect follower logs ...
  ... Passed --  28.6  5  1342  953354   102
Test (2B): RPC counts aren't too high ...
  ... Passed --   3.4  3    30    9050    12
PASS
ok      raft    58.142s


real    0m58.475s
user    0m2.477s
sys     0m1.406s
$
```

The "ok raft 58.142s" means that Go measured the time taken for the 2B tests to be 58.142 seconds of real (wall-clock) time. The "user 0m2.477s" means that the code consumed 2.477 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 2B 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.

Handin procedure for lab 2B

First, double-check that your code passes the 2B tests, **and** still passes the 2A tests. Then, run `make lab2b` to upload your code to the [submission site](#).

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 are logged in, which can be used to upload the lab from the console as follows.

```
$ cd ~/6.824
$ echo "XXX" > api.key
$ make lab2b
```

Part 2C

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.

A real implementation would write Raft's persistent state to disk each time it changed, and would read the 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; see the comments in `persist()` and `readPersist()`. `labgob` is like Go's `gob` encoder but prints error messages if you try to encode structures with lower-case field names.

> **TASK**
>
> Insert calls to `persist()` at the points where your implementation changes persistent state. Once you've done this, you should pass the remaining tests.

> **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:** Run `git pull` to get the latest lab software.
- **Hint:** Many of the 2C tests involve servers failing and the network losing RPC requests or replies.
- **Hint:** You will probably need the optimization that backs up nextIndex by more than one entry at a time. Look at 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 6.824 Raft lectures.

- **Hint:** A reasonable amount of time to consume for the full set of Lab 2 tests (2A+2B+2C) is 4 minutes of real time and one minute of CPU time.

Your code should pass all the 2C tests (as shown below), as well as the 2A and 2B tests.

```
$ go test -run 2C
Test (2C): basic persistence ...
  ... Passed --   7.2  3  206   42208    6
Test (2C): more persistence ...
  ... Passed --  23.2  5 1194  198270   16
Test (2C): partitioned leader and one follower crash, leader restarts ...
  ... Passed --   3.2  3   46   10638    4
Test (2C): Figure 8 ...
  ... Passed --  35.1  5 9395 1939183   25
Test (2C): unreliable agreement ...
  ... Passed --   4.2  5  244   85259  246
Test (2C): Figure 8 (unreliable) ...
  ... Passed --  36.3  5 1948 4175577  216
Test (2C): churn ...
  ... Passed --  16.6  5 4402 2220926 1766
Test (2C): unreliable churn ...
  ... Passed --  16.5  5  781  539084  221
PASS
ok      raft    142.357s
$
```

Handin procedure for lab 2C

First, double-check that your code passes all the 2A, 2B, and 2C tests. Then, run `make lab2c` to upload your code to the [submission site](#).

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 are logged in, which can be used to upload the lab from the console as follows.

```
$ cd ~/6.824
$ echo "XXX" > api.key
$ make lab2c
```

Please post questions on [Piazza](#).