Project 3: Raft Leader Election

Due May 25 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip and gz
Available May 8 at 12am - May 26 at 11:59pm 19 days

This assignment was locked May 26 at 11:59pm.

Introduction

In this lab, you'll build Raft, a replicated state machine protocol. A replicated service achieves fault tolerance by storing complete copies of its state on multiple replica servers. Replication allows the service to continue operating even if it experiences failures in some of its servers. Here arise challenges in that some of the replicas may be holding different copies of data.

Raft manages a service's state replicas and helps the service sort out what the correct state is after failures by implementing 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 will implement Raft as a Go object type and associated methods, 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 consult the **extended Raft paper** **(https://pdos.csail.mit.edu/6.824/papers/raft-extended.pdf)** and the Raft lecture notes. You may find it useful to look at this **illustration** **(http://thesecretlivesofdata.com/raft/)** of the Raft protocol and advice about **locking** **(https://pdos.csail.mit.edu/6.824/labs/raft-locking.txt)** and **structure** **(https://pdos.csail.mit.edu/6.824/labs/raft-structure.txt)** for concurrency. For a wider perspective, have a look at Paxos, Chubby, Paxos Made Live, Spanner, Zookeeper, Harp, Viewstamped Replication, and **Bolosky et al.** **(http://static.usenix.org/event/nsdi11/tech/full_papers/Bolosky.pdf)**

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).

Here is the skeleton code for this project: **project_3.tar.gz** . **Include the** raft **folder (inside the compressed file) within the src directory next to main and mapreduce.**

Your work: Over the course of the following exercises, you will have to write/modify following files yourself.

- raft.go

We have broken the lab in two parts which you will tackle in this and the next project (Project 4).

Project 3

We supply you with skeleton code and tests in *src/raft* and a RPC-like system in *src/labrpc*. You will implement Raft by writing code in ***raft/raft.go****,* expanding on the current skeleton code there. Your project must support the following interface, which the tester and your key/value server will use. You'll find more details in the comments within *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
```

To create a Raft peer, a service calls ***Make(peers, me, ...)***. The *peers* argument consists of an array of network identifiers of the Raft peers to use with the labrpc RPC. The *me* argument is the index of the current peer in the peers array. ***Start(command)*** tells Raft to commence the processing and to append the command to the replicated log. *Start()* should not wait for the log appends to complete, rather it should return immediately. Your implementation should send an ***ApplyMsg*** for each newly committed log entry to the *applyCh* argument of *Make()*.

The peers should exchange RPCs using the labrpc Go package that is provided. While it is modeled after Go's rpc library, it uses Go channels internally rather than sockets. *raft.go* contains some example code that sends an RPC (*sendRequestVote()*) as well as handling an incoming RPC (*RequestVote()*). You must use *labrpc* instead of Go's RPC package due to the fact that the *labrpc* is told by the tester to delay RPCs, re-order them, and delete them to simulate challenging network conditions under which your code should work correctly. We will test your code with the *labrpc* as was handed out, so you shouldn't make or rely on any changes to that code.

In this section, you will implement **leader election and heartbeats** (*AppendEntries* RPCs without log entries). The goal here is for a single leader to be elected, remain the leader if there are no failures, and to have a new leader elected if the old leader fails or if packets to/from the old leader are lost.
To test this part of the code run:

```
go test -run 3A
```

Hints and good ideas:

- *raft.go* can maintain any state you need in the *Raft* struct. A struct will also need to be defined in order to hold information about each log entry. Your code should, as closely as possible, follow Figure 2 in the paper.
- Fill in the structs *RequestVoteArgs* and *RequestVoteReply*. Create a background goroutine in *Make()* to periodically kick off leader election by sending out *RequestVote* RPCs when it hasn't heard from another peer for a while. This way, if there is already a leader the peer will learn about it, or become leader itself.
- Implement the *RequestVote()* RPC handler so that servers will vote for one another
- Define an *AppendEntries* RPC struct for heartbeats and send them out periodically from the leader. So that other servers don't step forward when one has already been elected, write an *AppendEntries* RPC handler method that resets the election timeout.
- Ensure that the election timeouts of different peers fire don't always fire at the same time. This keeps peers from voting for themselves which will keep any of them from being elected.
- Do not send out heartbeat RPCs from the leader more than ten times per second.
- Your Raft is required to elect a new leader within five seconds of the failure of the old leader. For this, you must pick election timeouts that are short enough that it's very likely for an election to complete in less that five seconds even if multiple rounds are required.
- While Section 5.2 of the paper notes that election timeouts range from 150 to 300 milliseconds, such a range only makes sense if heartbeats are sent out considerably more often than once per 150 milliseconds. You will have to use an election timeout larger than the paper's 150-300 ms as you are limited to 10 heartbeats per second. Do not make the time too large as you still must elect a leader within 5 seconds.
- Go's **rand** (https://golang.org/pkg/math/rand/) package may be useful.
- You'll find that you'll need to write code that takes actions periodically or after delays in time. ***time.Sleep()*** (https://golang.org/pkg/time/#Sleep) is useful for this, asise *time.Timer* or *time.Timer* although they are more difficult to use correctly.
- Go RPC only sends struct fields whose names begin with capital letters. Substructures must also have capitalized field names. The *labgob* package will warn you about this; don't ignore the warnings.

Be sure your implementation passes the (3A) tests before submitting. You must also ensure that your code is thread-safe. You can check this using the -race parameter when you run the testing code;

```
go test -race -run 3A
```

Submission instruction

File type: gz or zip

Filename: project3_YOUR_NAME (e.g. project3_Byungjin_Jun.tar.gz)

- **Only one person in a group** should submit your file. The filename should be the name of one of the group member.

File contents: ALL FILES of the project above the *src* level. (**don't modify other files, except for raft.go**)