Project 4: Raft Log Consensus

Due Jun 12 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip and gz
Available May 22 at 12am - Jun 13 at 11:59pm 23 days

This assignment was locked Jun 13 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** (same as the code base provided in project 3). 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, the first of which tackled in Project 3.

There is a third, NOT ASSIGNED, part that we describe in **the missing piece** below and which code - for completion - we provide you with in the tarball.

Project 4

It is necessary that Raft maintains a consistent, replicated log of operations. The *Start()* call at the leader begins the process of adding a new operation to the log; subsequently, the leader sends new operations to the other servers using *AppendEntries* RPCs.

In this part, you will implement the leader and follower code to append new log entries. You will implement ***Start()***, complete the ***AppendEntries* RPC structs**, send them, fill out the ***AppendEntry* RPC handler**, and advance the ***commitIndex*** of the leader. You should first pass the *TestBasicAgree3B()* test (in *test_test.go*). Then you should get all the 3B tests to pass:

```
go test -run 3B
```

Hints and good ideas:

- You must implement the election restriction (detailed in section 5.4.1 of the paper).
- Failing early 3B tests may stem from holding un-needed elections, such as an election even though the current leader is alive and communicating with all peers. This can prevent agreement in situations where the tester believes an agreement is possible. Not sending out heartbeats immediately after winning an election or bugs in election timer management can cause un-needed elections.
- You may need to wait for certain events to occur within your code. Do this with Go's channels, Go's **condition variables (https://golang.org/pkg/sync/#Cond)**, or (as a last resort) by inserting a *time.Sleep(10 * time.Millisecond)* in each loop iteration.
- Allow for extra time to rewrite your implementation with insight gained from structuring concurrent code

The 3B tests will be used to grade this assignment. 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 3B
```

The missing piece (extra credit: 20 points)

If a Raft-based server reboots, it should resume service where it left off. Therefore, it is necessary that Raft keep persistent state that survives a reboot. This is mentioned in Figure 2 of the paper, and *raft.go* contains examples of how to save and restore a persistent state.

In the real world, this would be done by writing Raft's persistent state to disk each time it changes and reading from the disk after a reboot. For this lab, you will instead use a *Persister* object (in *persister.go*). Whoever calls *Raft.Make()* provides a *Persister* that initially holds Raft's most recently persisted state, if there is any. Raft should initialize its state from the *Persister* and then use it to save its persistent state each time the state changes. For this, use the *Persister*'s *ReadRaftState()* and *SaveRaftState()* methods.

In this part, you will complete the functions ***persist()*** and ***readPersist()*** in *raft.go* by adding code to save and restore persistent state. You will have to encode (or "serialize") the array state as a byte array in order to pass it to the *Persister*. To do this, use the *labgob* encoder that we provide; see comments in *persist()* and *readPersist()*. *labgob* is inspired by Go's *gob* encoder; the only difference is that *labgob* checks if your encoding structures with lower-case field names.

You now must determine where in the Raft protocol your servers are required to persist their state and insert calls to *persist()* at those points. In *Raft.Make()* there is already a call to *readPersist()*. Once this is done you should pass the remaining tests. You may want to try to first pass the "basic persistence" test;

```
go test -run TestPersist13C
```

before tackling the remaining ones;

```
go test -run 3C
```

Submission instruction

File type: gz or zip

Filename: project4_YOUR_NAME (e.g. project4_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**)