Assignment 2: Raft Leader Election

Deadline: Sunday, 1st November 2020 at 11:59 PM

This is the first in a series of assignments in which you'll build a fault-tolerant key/value storage system. You'll start in this assignment by implementing the leader election features of Raft, a replicated state machine protocol. In Assignment 3 you will complete Raft's log consensus agreement features. You will implement Raft as a Go object with associated methods, available to be used as a module in a larger service. Once you have completed Raft, the course assignments will conclude with such a service: a key/value service built on top of Raft.

Note: you are required to submit **ONLY** your implementation of raft.

Note: This assignment is to be done **individually.**

Note: You should post any assignment related query ONLY on piazza. Do not directly email the course staff.

Note: Course policy about **plagiarism** is as follows:

- Students must not share actual program code with other students.
- Students must be prepared to explain any program code they submit.
- Students must indicate with their submission any assistance received.
- All submissions are subject to plagiarism detection.
- Students cannot copy code from the Internet.
- Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Raft Overview

The Raft protocol is used to manage replica servers for services that must continue operation in the face of failure (e.g. server crashes, broken or flaky networks). The challenge is that, in the face of these failures, the replicas won't always hold identical data. The Raft protocol helps sort out what the correct data is.

Raft's basic approach for this 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 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.

You should read up on the **raft paper uploaded on LMS. We have highlighted all relevant parts for this assignment,** please make sure to read the highlighted present till page 6. You may also find this illustrated Raft guide useful to get a sense of the high-level workings of Raft.

Software and Testing

Make sure to place src/raft and src/labrpc in your $GOPATH folder.

Before you have implemented anything, your raft tests will fail, but this behavior is a sign that you have everything properly configured and are ready to begin:

```
$ cd "$GOPATH/src/raft"
$ go test -run Election

Test: initial election ...
--- FAIL: TestInitialElection (5.00s)
config.go:286: expected one leader, got none
Test: election after network failure ...
--- FAIL: TestReElection (5.00s)
config.go:286: expected one leader, got none
FAIL
exit status 1
```

Your Task: Leader Election

For this assignment, we will focus primarily on the code and tests for the Raft implementation in **src/raft**. It is worth your while to read and digest the code in this package, primarily the **raft/raft.go** file, **after you have read this handout**. **The comments** in the code will help you a lot.

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

The natural first task is to 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 the election to work, you will also need to implement the **RequestVote()** RPC handler so that servers will vote for one another.

To implement heartbeats, you will need to define an **AppendEntries** RPC struct (though you will not need any real payload yet) and have the leader send heartbeats 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.

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 a leader.

Your Raft implementation must support the following interface, which the tester and (eventually) your key/value server will use. You'll find more details in the 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 peer 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).
// you do not need to use this struct for this assignment, but you should
// not tamper with it because the tester uses it
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 **(you don’t need to do this for this assignment). Start()** should return immediately, without waiting for this process to complete. (**Just need to do this)**

Your Raft peers should exchange RPCs using the labrpc Go package that we provide to you. It is modelled 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()**).

Implementing leader election and heartbeats (empty **AppendEntries** calls) should be sufficient for a single leader to be elected and -- in the absence of failures -- stay the leader. It should also redetermine leadership after the current leader fails. Once you have this working, you should be able to pass the two Election "go test" tests:

```
$ go test -run Election
Test: initial election ...
  ... Passed
Test: election after network failure ...
  ... Passed
PASS
  ok  raft7.008s
```

Resources and Advice

- 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.
- Your implementation should follow the paper's description closely since that's what the tests expect. **Figure 2 of the paper** also attached **in the appendix** may be useful as a pseudocode reference.
- Add **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.
- 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, tha**t the 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.
- The paper 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.
- You may find Go's rand useful.
- 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.
- Read the attached advice documents about locking and structure. **They will definitely help you get started in the right direction.**.
- If your code has trouble passing the tests, read the paper's Figure 2 again, which is **attached in Appendix**; the full logic for leader election is spread over multiple parts of the figure.
- Don't forget to implement **GetState().**
- 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.
- 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 Election > ou**t. 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.

- 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).
- Check your code with **go test -race**, and fix any race cases it reports

Evaluation

You can earn up to 20 points from this assignment. There is no extra credit or bonus.

1. **InitialElection -** 5 Points
2. **ReElection -** 5 Points
3. **No Race Conditions** - 5 points
4. **Go Formatting** - 1 point
5. **Manual Grading -** 4 points
   You can obtain full points from this if you satisfy these conditions:
   - Your submission only uses the permitted packages.
   - Your submission has good coding style that adheres to the formatting and naming conventions of Go.

Submission

Submit the **raft.go** file on LMS after renaming it to **raft_<R>.go**, where **<R>** should be replaced by your roll number, e.g. **raft_17030010.go.** Please submit your code with all print statements removed.

Late Submissions

You should submit your assignment on LMS before its due time.

If you submit your work late, we will award you a fraction of the score you would have earned on the assignments had it been turned in on time, according to this sliding scale:

- 90% for work submitted up to 24 hours late
- 80% for work submitted up to 2 days late
- 70% for work submitted up to 3 days late
- 60% for work submitted up to 5 days late
- 50% for work submitted after 5 days late

For example, if you should have earned 8/10 points but submitted 36 hours late, you will instead earn 6.4 points.

Besides the above policy, **you are allowed five "free" late days during the semester** (that can be applied to one of assignments 1 through 3; the final assignment will be due tentatively on final day of classes [12 Dec] and cannot be turned late. **The last day to do any late submission is also the final day of classes, even if you have free late days remaining**). You do not need to tell us that you are applying your "late day" -- we'll remove the late penalty at the end of the semester from the assignment that benefits you the most.

Appendix

State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

Figure 2: A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction). The server behavior in the upper-left box is described as a set of rules that trigger independently and repeatedly. Section numbers such as §5.2 indicate where particular features are discussed. A formal specification [31] describes the algorithm more precisely.