Project 3: Raft

1 Overview

Important Dates:

Project release: **Friday, November 17, 2017 at 02:00pm**
Checkpoint due: **Wednesday, November 29, 2017 at 11:59pm**
Final Test due: **Friday, December 8, 2017 at 11:59pm**
Submission limits: **15 Autolab submissions per checkpoint**

In this project you'll implement the Raft algorithm, a replicated state machine protocol. There will be one checkpoint for the implementation of your Raft. Keep in mind that there will be no office hours over Thanksgiving, so you will want to start early. For more information regarding what portion of the project is expected to be completed for the checkpoint and the final test, please refer to section 4 and 5.

The starter code for this project is hosted as a read-only repository on GitHub. For instructions on how to build, run, test, and submit your server implementation, see the `README.md` file in the project's root directory. To clone a copy, execute the following Git command:

```
git clone https://github.com/CMU-440-F17/p3.git
```

You must work on this project **individually**. As per course policy, you may turn the project in up to **two** days late, and each day will incur a 10% deduction from your score.

2 Raft

A replicated service (e.g., key/value database) achieves fault tolerance by storing copies of its 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 and ordering of the entries in the log. Each replica asynchronously executes the client requests 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 the log of the recovered server 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 project 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: Only RPC may be used for interaction between different Raft instances. For example, different instances of your Raft implementation are not allowed to share Go variables. Your implementation should not use files at all.

In this project you'll implement a part of the Raft design described in the extended paper. You will not implement persistence, cluster membership changes (Section 6) or log compaction / snapshotting (Section 7).

You should consult the extended Raft paper. You may find it useful to look at this illustrated guide to Raft. For a wider perspective, have a look at Paxos, Chubby, Paxos Made Live, Spanner, Zookeeper, Harp, Viewstamped Replication, and Bolosky et al.

Hint: 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.

Hint: Read and understand the extended Raft paper before you start. Your implementation should follow the paper's description closely, particularly Figure 2, since that's what the tests expect.

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.

3 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 will use. You'll find more details in comments in `raft.go`.

```
// create a new Raft server instance:
rf := Make(peers, me, 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 tester via the |applyCh| passed to
// |Make|.
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. `Start()` should return immediately, without waiting for this process to complete. The service expects your implementation to send an `ApplyMsg` for each new 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 modify `labrpc` because we will test your code with the `labrpc` as handed out. **Note:** The `labrpc` package only provides a subset of the functionality of Go's RPC system. For instance, asynchronous RPC calls are not provided by `labrpc`.

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.

4 Checkpoint

4.1 Task

Implement leader election and heartbeats (`AppendEntries` RPCs with no log entries). The goal for checkpoint 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 3A` to test your checkpoint code.

Be sure you pass the checkpoint tests before submitting. Note that the checkpoint tests test the basic operation of leader election. The final tests will test leader election in more challenging settings and may expose bugs in your leader election code which the checkpoint tests miss.

4.2 General Guidelines

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.

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.

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.

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.

The tester requires that the leader send heartbeat RPCs no more than ten times per second.

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.

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.

4.3 Notes and Hints

There are some details and hints we want to emphasize here to help you pass our tests:

- Go RPC sends only those struct fields whose names start with capital letters. Substructures must also have capitalized field names (e.g. fields of log records in an array). Forgetting to capitalize field names sent by RPC is the single most frequent source of bugs while using RPCs.
- You may find Go's time.Sleep() and 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().
- 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.
- 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 3A > 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.
- You should check your code with `go test -race`, and fix any races it reports.

5 Final Test

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.

5.1 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 `TestBasicAgree()` test (in `test_test.go`). Once you have that working, you should get all the final tests to pass (`go test -run 3B`).

5.2 General Guidelines

While the Raft leader is the only server that initiates appends of new entries to the log, all the servers need to independently give each newly committed entry to their local service replica (via their own `applyCh`). You should try to keep the goroutines that implement the Raft protocol as separate as possible from the code that sends committed log entries on the `applyCh` (e.g., by using a separate goroutine for delivering committed messages). If you don't separate these activities cleanly, then it is easy to create deadlocks. Without a clean separation, a common deadlock scenario is as follows: an RPC handler sends on the `applyCh` but it blocks because no goroutine is reading from the channel (e.g., perhaps because it called `Start()`). Now, the RPC handler is blocked while holding the mutex on the Raft structure. The reading goroutine is also blocked on the mutex because `Start()` needs to acquire it. Furthermore, no other RPC handler that needs the lock on the Raft structure can run.

You will need to implement the election restriction (section 5.4.1 in the paper).

6 Hand In

Create a tarball of the `raft/` folder with the `raft.go` file in it.
Use the following command:
`tar cvf raft.tar raft/`

7 Project Requirements

As you write code for this project, also keep in mind the following requirements:

- You must work on this project individually. You are free to discuss high-level design issues with other people in the class, but every aspect of your implementation must be entirely your own work.

- You must format your code using `go fmt` and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.
- You may use any of the synchronization primitives in Go's `sync` package for this project.