Project 2: The Raft Consensus Algorithm

Please read all pages before writing code.

1 Overview

Important Dates:

Project release: **Monday, October 18, 2021**
Checkpoint due: **Saturday, November 6, 2021 at 11:59pm**
Full project due: **Saturday, November 13, 2021 at 11:59pm**
Submission limits: **15 Gradescope submissions per checkpoint**

In this project, you'll implement the Raft consensus algorithm, a replicated state machine protocol. 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 sections 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-F21/P2.git
```

You must work on this project **individually**. You will have 15 submissions for each due date. The late-day policy of previous projects will apply here as well. No submissions will be accepted beyond two days past each deadline. Your Gradescope submission will output a message showing how many submissions you have made and how many you have left. Gradescope will allow you to submit beyond this limit, but we will be checking manually.

There will be no manual style grading for the checkpoint. However, 4 points are allocated for manual style grading on the final submission. Specifically, we are looking for good function headers, comments for variables, and no debugging print statements or chunks of dead code. You're allowed to use mutexes for this project (see Project Requirements).

2 Raft

A replicated service (e.g., key/value database) achieves fault tolerance by storing copies of its data on multiple replicas. Replication allows the service to continue operating even if some of its replicas experience failures (crashes or a broken/flaky network). The challenge is that failures may cause the replicas to hold differing copies of the data.

One protocol to ensure all of these copies of the data are consistent across all non-faulty replicas is Raft. Raft implements a replicated state machine by sequencing client requests into a log, and ensuring that the replicas agree on the contents and ordering of the log entries. Each replica *asynchronously* applies the client requests in the order they appear in the replica's log of the service's state. If a replica fails and later recovers, Raft takes care of bringing the log of the recovered replica up to date. Raft will continue to operate as long as at least a quorum of replicas is alive and able to communicate. If a quorum is not available, Raft will stop making progress but will resume as soon as a quorum becomes available.

In this project, you will implement Raft in Go. Your Raft module will implement a replicated log using the Raft protocol, using RPC to communicate between replicas. Your implementation should support an indefinite sequence of numbered commands (log entries). Each log entry is comprised of a client command and an index number. After a log entry is committed, Raft will "apply" the log entry by sending the committed log entry to the application that is using your Raft module.

Note: Only RPC may be used for interaction between different Raft instances. For example, different peers in your Raft implementation are not allowed to share Go variables or access shared files/sockets. Communicating between 2 replicas using anything except the approved `rpc` package will result in losing 50% of the points on this project.

You'll implement a part of the Raft protocol described in the extended paper. You do not need to implement persistence, cluster membership changes (Section 6) or log compaction / snapshotting (Section 7).

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

Tip 1: Start early. Although you can implement Raft in a relatively small number of lines of code (our reference solution is $\approx$ 700 lines), getting it to work correctly will be challenging. Both the algorithm and the code are 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.

Tip 2: Read and understand the extended Raft paper before you start. Your implementa-

tion should follow the paper's description closely, particularly Figure 2, since that's what the tests expect. Figure 2 is reproduced on the last page of this handout.

Tip 3: Your first implementation may not be clean enough to easily reason about its correctness. Give yourself enough time to rewrite your implementation at least once.

3 The code

Implement Raft by adding code to `raft/raft.go`. In that file, you'll find a bit of skeleton code and 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`.

```
// rf = NewPeer(...)
//   Create a new Raft peer.
//
// rf.PutCommand(command interface{}) (index, term, isleader)
//   Start agreement on a new log entry
//
// rf.GetState() (me, term, isLeader)
//   Ask a Raft peer for "me" (see line raft.go:75),
//   its current term, and whether it thinks it is a leader
//
// ApplyCommand
//   Each time a new entry is committed to the log, each Raft peer
//   should send an ApplyCommand message to the service (e.g. tester) on the
//   same server, via the applyCh channel passed to NewPeer()
```

A service calls `NewPeer(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. `PutCommand(command)` asks Raft to start the processing to append the command to the replicated log. `PutCommand()` should return immediately, without waiting for this process to complete. The service expects your implementation to send an `ApplyCommand` for each new committed log entry to the `applyCh` argument to `NewPeer()`.

Your Raft peers should exchange RPCs using the `rpc` Go package that we provide to you (`https://github.com/CMU-440-F21/P2/src/github.com/cmu440/rpc`). 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 `rpc` instead of Go's RPC package is that the tester tells `rpc` to delay RPCs, re-order them, and delete them to simulate challenging network conditions under which your code should work correctly. Any modifications you make to `rpc` will be discarded before grading.

Note: The `rpc` package only provides a subset of the functionality of Go's RPC system. For instance, asynchronous RPC calls are not provided by `rpc`.

4 Checkpoint

4.1 Task

Implement leader election and heartbeats (`AppendEntries` RPCs with no log entries). The goal for the 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 -race -run 2A` 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 under 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 `NewPeer()` 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 the leader is, 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 a `AppendEntries` RPC struct (though you may not need all the arguments yet), and have the leader send them out periodically. Write a `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. *DO NOT USE THEIR RECOMMENDED TIMEOUT INTERVALS.* Those values only make sense if you use a heartbeat interval significantly smaller than 150 ms (the lower bound), which is not the case in this project. Instead, here are some general guidelines on how to choose intervals in this project:

1. You should not send heartbeat RPCs more than 10 times per second.
2. It is not recommended to use Go's native rpc package, instead use our own.
3. Election timeout should be multiple times greater than your heartbeat interval.
4. Election timeout should be small enough to allow re-election of a leader under 5 seconds.
5. Make sure the election timeouts in different peers don't always fire at the same time, as explained earlier.
6. We recommend you run test > 10 times locally to make sure you do not have concurrency issues before you submit to Gradescope.

4.3 Notes and Hints

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

- Go RPC only serializes struct fields with capitalized names. Sub-structures 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.

- Unlike in P1, you may use buffered channels of any size in this project.
- 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 debug logs when a peer sends or receives a message and redirecting it to a file (`go test -race > out.txt`) and examining the file to trace the execution of your system.
- You should check your code with `go test -race`, and fix any races it reports.
- You should try your code with varying numbers of CPU cores (`go test -race -cpu=N`)
- We provide a simple logging framework in the starter code to allow you to keep separate debug logs for each peer.

4.4 Debugging

In the starter code, we provide you a simple logging framework to allow you to maintain separate debug logs for for each peer, with the option to output to files or stdout. You are welcome to use, modify, remove, or ignore this logging code – however, we expect you to have clear, understandable debug log files before asking questions at office hour or on Piazza. The easiest way to do this is to use our included debug logging framework.

These debug logs can easily be disabled by setting

```
kEnableDebugLogs = false
```

at the top of raft.go. Be sure to disable your debug logs before submitting to Gradescope.

You can output your debug log output to stdout by setting

```
kLogToStdout = true
```

at the top of raft.go. Each log line will be prefixed by the peer's unique name.

If you set kLogToStdout to false, each peer's log will instead be output into a separate .txt file in the directory described by `kLogOutputDir`. By default, we enable debug logs and output them to stdout.

5 Final Test

We want Raft to keep a consistent, replicated log of operations. A call to `PutCommand()` at the leader starts the process of adding a new operation to the log; the leader sends the new operation to the other servers using `AppendEntries` RPCs.

5.1 Task

Implement the leader and follower code to append new log entries. This will involve implementing `PutCommand()`, 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 -race -run 2B`).

5.2 General Guidelines

While the Raft leader is the only server that initiates appends of new entries to the log, all the replicas need to independently give each newly committed log entry to their local service replica (via their `applyCh` passed to NewPeer()). 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 `PutCommand()`). Now, the RPC handler is blocked while holding the mutex on the Raft structure. The reading goroutine is also blocked on the mutex because `PutCommand()` 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

Reminder: Please disable or remove **all** debug prints regardless of whether you are using our logging framework or not before submitting to Gradescope. This helps avoid inadvertent failures and messy autograder outputs and style point deduction.

For both the checkpoint and the final submission, create `handin.zip` using the following

command under the `P2/` directory, and then upload it to Gradescope.
`sh make_submit.sh`

7 Testing and Grading

We will use Gradescope to automatically grade your implementation for correctness, and manual grading for style. In addition to tests we provide you in `raft_test.go`, we will run additional, more extensive tests on Gradescope for both your checkpoint and the final submissions. We will not be able to provide you any details (except for the `stdout` you'll see on Gradescope) about any of these new tests. We will run each test multiple times – you should pass every invocation of a test to pass that test.

You are encouraged to write new tests on your own. You are, however, not required to do so and will not be graded on any new tests you write. Here are some tips to do this. Refer to `raft_test.go` as you read these tips.

- See `TestInitialElection2A`: you can use `cfg.checkOneLeader()` to check for a leader's election and to get the current leader's ID.
- See `TestFailAgree2B`: you can use `cfg.one(value, num_servers)` to start an agreement.
- See `TestFailNoAgree2B`: you can use `cfg.disconnect(server_id)` and `cfg.connect(server_id)` to disconnect and connect servers. You can also directly call `PutCommand()` on one of the Raft peers by using `cfg.rafts`.

The checkpoint (Part A) is worth 45 points. The following table shows the point distribution for the final version of the project:

Part A	45 points
Part B	145 points
Manual Grading of Part B	4 points
go fmt of Part B	1 point
Total	195 points

Note that the tests for Part A are run and graded for points for both the checkpoint and the final version of the project. However, there is no manual or style grading on Part A as part of the checkpoint or the final.

8 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 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.
- For the tester to function correctly, please use the provided `rpc` package instead of Go's native one.

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.