MP2 — Leader Election and Log Consensus with Raft

Due: 11:59 p.m., **Wednesday, April 5th**

Notes

- You are welcome to switch groups from MP1.
- This assignment has a very different structure compared to MP0 and MP1. You do not need to run your distributed system on a cluster of VMs. Instead, in this assignment, you are provided with an emulation framework that runs on a single machine. Different processes communicate with each other over an emulated network.
- Your goal is to implement Raft's leader election and log consensus logic by beefing up a skeleton code that we have provided.
- We also provide you with a test suite that emulates different scenarios to stress test your code (e.g. processes crashing or getting disconnected from one another). We will use the same test suite to evaluate your MP.
- The emulation framework and test harness are written in Go, and therefore you **must** use Go for this MP.

Overview

In this MP you'll implement Raft, a replicated state machine protocol. A replicated service achieves fault tolerance by storing complete copies of its state (i.e., 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 organizes client requests into a sequence, called the log, and ensures that all the replica servers see the same log. Each replica executes client requests in log order, applying them to its 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. 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 MP you'll implement Raft as a Go object type with associated methods, which is meant to be used as a module in a larger service (and for this MP, is used by a tester provided to you). A set of Raft instances talk to each other via RPCs (remote procedure calls) 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 follow the design in the `extended Raft paper`, with particular attention to Figure 2. You'll specifically implement the logic for leader election and log consensus. We will not test scenarios where a crashed process is restarted again, so you need not handle persistent state. However, we will test scenarios where a process may become unreachable, and then the connection is established again. You will also *not* implement cluster membership changes (Section 6) and log compaction / snapshotting (Section 7). We have split this MP into two parts for your ease of implementation and testing: 2A (leader election) and 2B (log consensus). Your final submission should pass the test cases for both parts.

Collaboration Policy

As with all assignments, this work should be entirely your own (and your partner's). You are not allowed to look at anyone else's solution. Additionally, *you are not allowed to look at other Raft implementations that might be available online*. You may discuss the MP with other students, but you may not look at or copy anyone else's code or allow anyone else to look at your code. Please do not publish your code or make it available to current or future ECE428/CS425 students. Github repositories are public by default, so please don't put your code there unless you make the repository private. You can instead use a private git repository hosted by the Engineering GitLab.

Getting Started

The source code for this MP is available `here`. We supply you with skeleton code `src/raft/raft.go`. We also supply a set of tests, which you should use to drive your implementation efforts, and which we'll use to grade your submission. The tests are in `src/raft/test_test.go`. To get up and running, download the source code and execute the following commands.

```
$ cd src/raft
$ go test
Test (2A): initial election ...
--- FAIL: TestInitialElection2A (5.12s)
    config.go:282: expected one leader, got none
Test (2A): election after network failure ...
--- FAIL: TestReElection2A (4.98s)
    config.go:282: expected one leader, got none
Test (2B): basic agreement ...
--- FAIL: TestBasicAgree2B (10.00s)
    config.go:427: one(100) failed to reach agreement
…..
```

The tests obviously fail as Raft's replicated log consensus has not yet been implemented. You need to implement it by adding code to `raft/raft.go`. In that file you'll find 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, 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
```

A service calls `Make(peers,me,…)` to create a Raft peer (or process). The `peers` argument is an array of network identifiers of the Raft peers (including this one), for use with RPC. 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 the log appends to complete. The service expects your implementation to send an `ApplyMsg` for each newly committed log entry to the `applyCh` channel argument to `Make()`. The Raft peers communicate via RPCs. `raft.go` contains example code that sends an RPC (`sendRequestVote()`) and that handles an incoming RPC (`RequestVote()`). Your Raft peers should exchange RPCs using the labrpc Go package (source in `src/labrpc`). The tester can tell `labrpc` to delay RPCs, re-order them, and discard them to simulate various network failures. While you can temporarily modify `labrpc`, make sure your Raft works with the original `labrpc`, since that's what we'll use to test and grade your MP. Your Raft instances must interact only with RPC; for example, they are not allowed to communicate using shared Go variables or files.

Part 2A: Leader Election

Implement Raft leader election and heartbeats (`AppendEntries` RPCs with no log entries). The goal for Part 2A 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 2A` to test your 2A code.

General hints and tips :

- You can't easily run your Raft implementation directly; instead you should run it by way of the tester, i.e. `go test -run 2A`.
- Follow the paper's Figure 2. At this point you care about sending and receiving RequestVote RPCs, the Rules for Servers that relate to elections, and the State related to leader election.
- Add the 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.
- 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.
- Don't forget to implement `GetState()`, which is what the tester uses to check for leaders.
- 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.
- Reading this advice about [`locking`] might be helpful. (We shared a similar advice earlier – this one has some relevant examples specifically in the context of this MP).

Useful Go features:

- 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.
- Some other Go features that you *may* find useful include: `select`, `conditional waits`, and Go `channels`.

Tips on testing and debugging your code:

- 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 2A > 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.
- 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). The `labgob` package will warn you about this; don't ignore the warnings.
- Check your code with `go test -race`, and fix any races it reports.

Be sure you pass the 2A tests before proceeding to the next part. *The tests might be non-deterministic – certain failure conditions may arise based on specific timing of events. Make sure you run the tests at least 5-6 times to evaluate whether your code passes them.* When you pass your test for 2A, you should see an output that looks something like this:

```
$ go test -run 2A
Test (2A): initial election ...
  ... Passed --   4.0  3   32    9170    0
Test (2A): election after network failure ...
  ... Passed --   6.1  3   70   13895    0
PASS
ok      raft    10.187s
$
```

Each "Passed" line contains five numbers; these are the time that the test took in seconds, the number of Raft peers (usually 3 or 5), the number of RPCs sent during the test, the total number of bytes in the RPC messages, and the number of log entries that Raft reports were committed. Your numbers will differ from those shown here. You can ignore the numbers if you like, but they may help you sanity-check the number of RPCs that your implementation sends. The grading script will fail your solution if it takes more than 600 seconds for all of the tests (`go test`), or if any individual test takes more than 120 seconds.

Part 2B: Log Consensus

Implement the leader and follower code to append new log entries, so that the `go test -run 2B` tests pass. Hints and Tips:

- Your first goal should be to pass `TestBasicAgree2B()`. Start by implementing `Start()`, then write the code to send and receive new log entries via `AppendEntries` RPCs, following Figure 2.
- You will need to implement the election restriction (section 5.4.1 in the paper).
- Note that there are implementation specific aspects you will need to think through, that Figure 2 (and the paper) do not cover in detail.
- One way to fail to reach agreement in the early 2B tests is to hold repeated elections even though the leader is alive. Look for bugs in election timer management, or not sending out heartbeats immediately after winning an election.
- Make sure you handle heartbeats (empty AppendEntries) with care. Ideally, your code should not have a special handler for heartbeats, and should ensure that all checks for `AppendEntries` are satisfied before accepting the RPC (or returning success).
- Another reason for failing some of the later tests is that it takes your implementation too long to elect a leader with processes staying stuck in the candidate state. To avoid this, make sure you are still incrementing the current term of the Raft peers correctly when you implement election restriction.
- To pass the final test in 2B, you may need to implement the optimization mentioned in Sec 5.3 of the paper (towards the end of page 7 and beginning of page 8).
- If your code has loops that repeatedly check for certain events, don't have these loops execute continuously without pausing, since that will slow your implementation enough that it fails tests. Use Go's `conditional waits`, or insert a `time.Sleep(10 * time.Millisecond)` in each loop iteration.

You can also check how much real time and CPU time your solution uses with the time command. Here's a typical output:

```
$ time go test -run 2B
Test (2B): basic agreement ...
  ... Passed --   1.6  3   18    5158    3
Test (2B): RPC byte count ...
  ... Passed --   3.3  3   50  115122   11
Test (2B): agreement despite follower disconnection ...
  ... Passed --   6.3  3   64   17489    7
Test (2B): no agreement if too many followers disconnect ...
  ... Passed --   4.9  5  116   27838    3
Test (2B): concurrent Start()s ...
  ... Passed --   2.1  3   16    4648    6
Test (2B): rejoin of partitioned leader ...
  ... Passed --   8.1  3  111   26996    4
Test (2B): leader backs up quickly over incorrect follower logs ...
  ... Passed --  28.6  5 1342  953354  102
Test (2B): RPC counts aren't too high ...
  ... Passed --   3.4  3   30    9050   12
PASS
ok      raft    58.142s

real    0m58.475s
user    0m2.477s
sys     0m1.406s
$
```

The “ok raft 58.142s” means that Go measured the time taken for the 2B tests to be 58.142 seconds of real (wall-clock) time. The “user 0m2.477s” means that the code consumed 2.477 seconds of CPU time, or time spent actually executing instructions (rather than waiting or sleeping). If your solution uses much more than a minute of real time for the 2B tests, or much more than 5 seconds of CPU time, you may run into trouble later on. Look for time spent sleeping or waiting for RPC timeouts, loops that run without sleeping or waiting for conditions or channel messages, or large numbers of RPCs sent.

As with 2A, the tests can be non-deterministic. Make sure you pass all of them at least 5-6 times.

Submission instructions

Your code (the single file `raft.go`) needs to be in a git repository hosted by the Engineering GitLab. Please disable all print statements (in case you are not using `DPrintf` from `util.go`), to ensure that your code does not print anything to std:out while testing, other than the test results. Failure to do so will result in a score penalty. We will be using one of the VMs on the campus cluster to test your code. Make sure your code passes the provided test cases on one of the campus VMs.

After creating your repository, please add the following NetIDs as members with *Reporter* access:

- gmk6
- jw22
- eashang2
- sm106
- radhikam
- ayxu2
- hanbog2
- srm14

You will also need to submit a report through Gradescope. In your report, you must include the URL of your repository and the full revision number (git commit id) that you want us to grade, as shown by `git rev-parse HEAD`. Make sure that revision is `push`ed to GitLab before you submit. Only one submission per group – it is set up as a “group submission” on Gradescope, so please make sure you add your partner to your Gradescope submission. In addition to this, your report should include the names and NetIDs of the group members. That’s it for this MP!

High-Level Rubric

- Test (2A) [40 points]
  - Test (2A): initial election [15 points]
  - Test (2A): election after network failure [25 points]
- Test (2B) [70 points]
  - Test (2B): basic agreement [10 points]
  - Test (2B): RPC byte count [5 points]
  - Test (2B): agreement despite follower disconnection [10 points]
  - Test (2B): no agreement if too many followers disconnect [10 points]
  - Test (2B): concurrent Start()s [10 points]

    - Test (2B): rejoin of partitioned leader [10 points]
    - Test (2B): leader backs up quickly over incorrect follower logs [10 points]
    - Test (2B): RPC counts aren't too high [5 points]

Note that failing an earlier test might result in cascading failures for follow-up tests. You should strive to pass the tests in the above order to score maximum points. To tackle potential non-determinism, we will run your code 5-6 times while grading – you will score the allocated points for a test only if it is successful each time we run your code.

Acknowledgement

We are thankful to the 6.824 course staff at MIT for this assignment.