Project 3: Raft

Due: 11:59 PM, April 14, 2022

Contents

1 Getting Started

Remember, if you write code on a department machine, you must use `go1.17` instead of just `go` (i.e. `go1.17 install` or `go1.17 test`). Add `alias go=go1.17` to your `~/.bashrc` for convenience.

```
git clone github.com/brown-csci1380-s22/tapestry-<TeamName>
```

Use the command above to get your repo from GitHub Classroom. Fix the import path in all files in the `cmd` directory, `client` directory, and `raft/cluster.go`. Use `go1.17 install ./...` to fetch all dependencies. Before you get started, please make sure you have read over, set up, and understand all the support code.

We highly encourage you to work in groups of two, but we understand that in some situations a group of three may be necessary. If you work in a group of three, you must implement an additional feature. These can be found in the Extra Credit section at the end of the handout. If you work in a group of three, you must contact the TAs and let them know you intend to work in a group of three, and if you will be implementing additional features.

Working alone is not allowed for this project. If you do not have a partner for any reason, please attempt to find one through the EdStem partner search functionality. If this is not successful, please email the HTAs for assistance.

2 Introduction

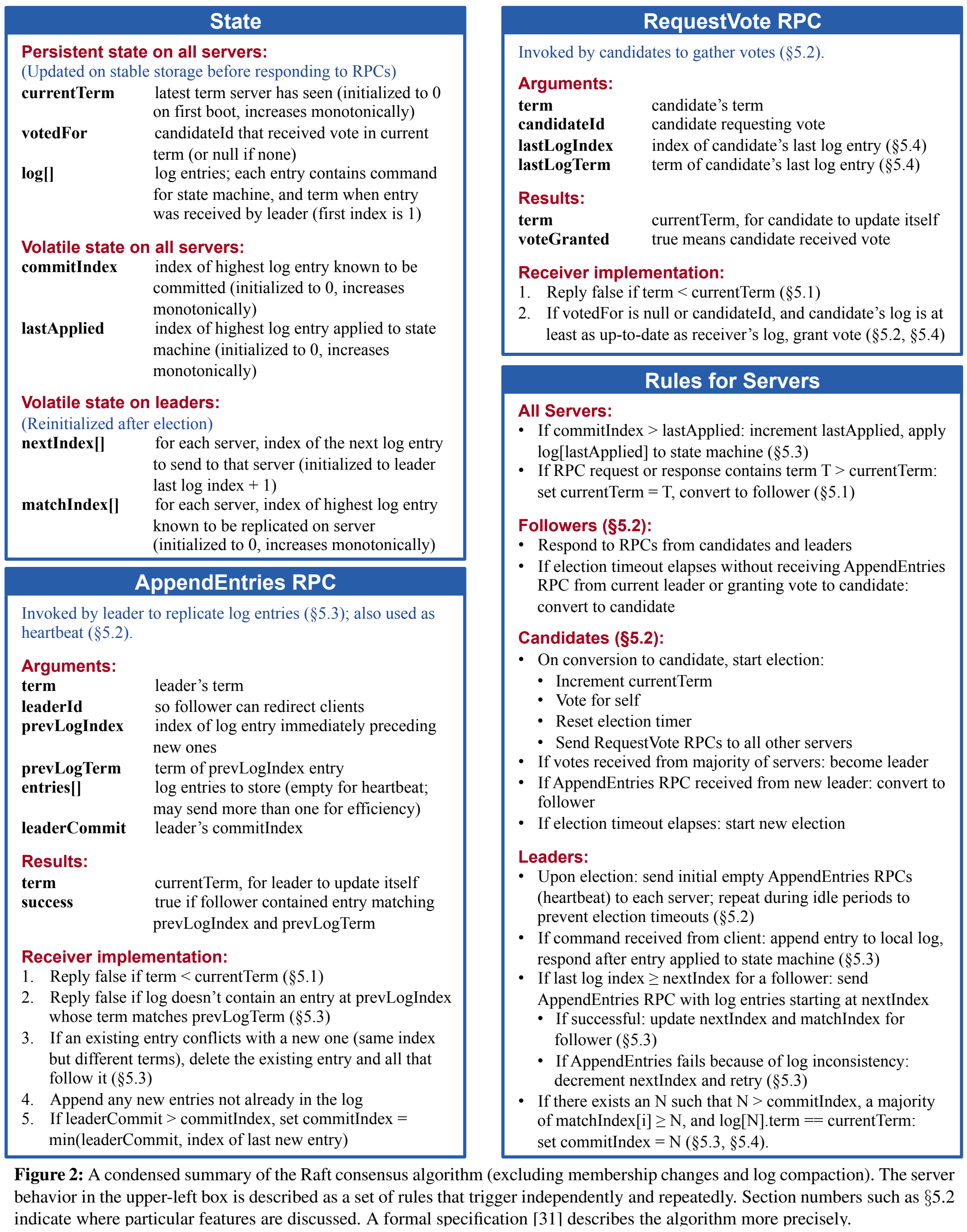
An important part of creating total-ordered, fault-tolerant distributed systems is providing the ability for multiple nodes to come to a consensus about state.

The problem of distributed consensus has been around for a long time and has typically been solved using implementations of the popular Paxos algorithm, which was initially published in 1998. Paxos, however, has been shown to be difficult to fully understand, let alone implement. The difficulties related to Paxos have spawned much work over the years in trying to make a more practical protocol.

In this spirit, a group of researchers at Stanford (Diego Ongaro and John Ousterhout) developed the Raft protocol in 2014, which is what you will be implementing in this project. Raft is a consensus protocol that was designed with the primary goal of understandability without compromising on correctness or performance (when compared to protocols like Paxos).

For this project, the Raft paper will serve as the **central source of truth**. All implementation details are found in the paper and this handout will simply refer to relevant sections in the paper for your guidance.

This project is also meant for you to experience what it's like to implement an algorithm directly described from a paper. Raft is great for this in that it is very thorough in describing all the details necessary for its implementation. We hope that this skill will be useful for you in your future CS career.



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.

Figure 1: This figure is from the Raft paper, and helps summarize a lot of the important details in the protocol. If you find a note particularly confusing, refer to the section specified (e.g., §3.5).

3 Raft Overview

The Raft protocol can be broken down into four major components that you will have to implement: Leader Election, Log Replication, Log Commitment, and Client Interaction.

For your reference we have included a cheat sheet summary of the consensus algorithm in Figure 1.

3.1 Leader Election

Leader election consists of a Raft cluster deciding which of the nodes in the cluster should be the leader of a given term. Refer to the cheatsheet, section **§5.2** and **§5.4.1** for implementation details.

3.2 Log Replication

Log replication consists of making sure that the Raft state machine is up to date across a majority of nodes in the cluster. Refer to the cheatsheet, section **§5.3**, **§5.4.2**, and **§5.5** for implementation details.

3.3 Log Commitment

Log commitment is responsible for ensuring consistency of data in the Raft cluster since leaders are allowed to overwrite follower logs. Refer to the cheatsheet, section **§5.3**, **§5.4.2**, and **figure 8** from the paper for implementation details.

3.4 Client Interaction

Client interaction consists of a node outside the Raft cluster making requests to modify the state machine of the cluster. Refer to the cheatsheet and section **§8** for implementation details.

4 Implementation

4.1 Project Layout

In the stencil code, you'll see three high level packages:

- `raft`: the core Raft protocol
- `hashmachine`: a state machine based on a hash chain
- `client`: a client API for the `raft` package

You'll see a `cmd` directory with three packages inside: `raft-cli`, which is a CLI for a Raft client (it uses the `client` package above) and `raft-node`, a CLI to create and control a Raft node, and `raft-cluster`, a CLI to simultaneously start a group of Raft nodes to test leader election. Note that the Raft client is specific to the hash chain implementation we provide.

4.2 Raft Protocol Implementation

You should implement the following functions:

- utils.go
  - `func randomTimeout(minTimeout time.Duration) <-chan time.Time`
- node_follower_state.go
  - `func (r *Node) doFollower() stateFunction`
  - `func (r *Node) handleAppendEntries(msg AppendEntriesMsg) (resetTimeout, fallback bool)`
- node_candidate_state.go
  - `func (r *Node) doCandidate() stateFunction`
  - `func (r *Node) requestVotes(electionResults chan bool, fallback chan bool, currTerm uint64)`
  - `func (r *Node) handleCompetingRequestVote(msg RequestVoteMsg) (fallback bool)`
- node_leader_state.go
  - `func (r *Node) doLeader() stateFunction`
  - `func (r *Node) sendHeartbeats() (fallback, sentToMajority bool)`

Each function that returns a `stateFunction` should contain the logic for the Raft node being in one of the three Raft states: Follower, Candidate, Leader. You can transition to another state by returning that state function. For example, `doLeader()` could be written to always transition to the Follower state:

When an RPC is received, the request is forwarded over a channel so that the function of the appropriate state can determine how it should be interpreted. For example, the following code would always reply successful to an AppendEntriesRPC:

The full list of channels you should handle are:

- `RaftNode.appendEntries`
- `RaftNode.requestVote`
- `RaftNode.registerClient`
- `RaftNode.clientRequest`

- `RaftNode.gracefulExit`

Feel free to add whatever support code you need. **DO NOT CHANGE THE PUBLIC APIs**.

4.3 CS 138 Specific Nuances

- `DefaultConfig()` specifies in-memory log storage by default. This means that if you shutdown your raft node, all of its logs and state will be lost. Raft requires persistence of logs and some state such as a node's last vote, so in-memory storage would violate those conditions. However, using in-memory storage will actually be incredibly convenient when you start testing your Raft implementation. When you would like to test a node failing and restarting, set the config's inMemory field to false.
- We use BoltDB, an embedded DB written in Go, as the underlying stable storage for Raft. BoltDB is widely used across the software industry and is featured in projects such as Consul, Hashicorp's service-discovery platform, and InfluxDB, a popular DB for metrics and analytics.
- A Raft node is uniquely identified by its listener port in this scheme. So if you start up a new node that has the same port as a node that was running before which had its state saved to disk, this new node will appear with the same state. To avoid this, you can either use in-memory storage, ensure that new nodes you start have distinct ports, or you can call `func (r *Node) RemoveLogs()` to remove the persisted data.

5 Testing

We expect to see several good test cases. This is going to be worth a portion of your grade. You can check your test coverage by using Go's coverage tool.

To aid you with testing, we have provided sample tests inside `example_client_test.go`, `example_election_test.go`, and `example_partition_test.go`. These are not exhaustive, but test some core components of your implementation. Consider them a starting point for devising more complete tests of your own.

We have also provided the helper functions used by our tests in `test_utils.go`. Feel free to use them within your own tests.

To help you test out the behavior of your implementation under partitions, we have provided you with a framework which allows you to simulate different network splits. You can find the relevant code in `network_policy.go`, and see how it can be used in `cmd/raft-node/main.go` and `example_partition_test.go`.

5.1 Building and Running

Once you're in your repo's higher level `raft` directory, to get Raft to build, you must first update your dependencies, like so:

```
$ go1.17 install ./...
```

Then, you can build and install our three binaries, `raft-node`, `raft-client`, and `raft-cluster` by running:

```
$ cd cmd
$ go install ./...
```

This generates three CLIs and places them in your `$GOPATH/bin`:

You are welcome to improve any of the CLIs as you see fit. For a list of available commands in each CLI, type `help` after entering.

- `raft-node`

  This is a CLI that serves as a console for interacting with Raft, creating nodes, and querying state.

  You can pass the following arguments to `raft-node`:

  - `-p <port>`: The port to start the server on. By default selects a random port.
  - `-c <addr>`: Address of an existing Raft node to connect to.
  - `-d=true`: Enable or disable debug. Default is false.
  - `-m=true`: Enable or disable in-memory store. Default is true.

- `raft-client`

  This is a sample client which allows you to connect to a Raft node and issue commands to the hash state machine.

  You can pass the following arguments to `raft-client`:

  - `-c <address>`: A raft node address for raft-client to connect to

- `raft-cluster`

  This is a CLI that will simultaneously start up a cluster of raft nodes and is meant primarily to test for leader-election

  You can pass the following arguments to `raft-cluster`:

  - `-n <number>`: The number of nodes in the raft cluster. Default is 3.
  - `-m=true`: Use in-memory storage for raft cluster. Default is true
  - `-d=true`: Enable or disable debug. Default is false

6 Style

You should use Go's formatting tool `gofmt` to format your code before handing in. You can format your code by running:

```
gofmt -w=true *.go
```

This will overwrite your code with a formatted version of it for all go files in the current directory. You will be graded for this!

7 Handing in

About Checkpoint: There is a checkpoint available on Gradescope. Make sure to submit in group with "add group member" in submission page. Your code will be run against a partial version of the test suite that will be used in final grading. The grader will return the log back with your results. Based on the log, you can make further improvements accordingly. You can submit to the checkpoint as many times as you would like up to the deadline.

You need to write a README documenting any bugs in your code, any extra features you added, and anything else you think the TAs should know about your project. Document the test cases you created and briefly describe the scenarios you covered.

When you are done, please submit your code to Gradescope under the correct assignment. We will use the last submission attempt for grading.

7.1 New for Raft

7.1.1 Test Case Descriptions

We will be releasing a list of *test case descriptions* for all the test cases shown on the checkpoint Gradescope assignment. These descriptions will be helpful in debugging why certain tests are failing. Stay tuned on Ed for an announcement on this document.

7.1.2 TA Check-Ins

We will be assigning each group a *mentor TA* who will set up a 15-minute check in that will count towards your final Raft grade. Before your check-in, here are some questions to ask yourself to prepare:

1. Did you read the Raft paper? Are there any questions you have about the consensus algorithm?
2. Do you have any questions about the stencil?

During the check-in, we will be looking for the following:

1. Please complete at least one of the four components of the consensus algorithm in 3 and be prepared to discuss your approach and demonstrate that this is working (passing a test on Gradescope, a test you wrote, a sample test, etc.).
2. We will be releasing an example scenario involving some number of Raft servers. Be prepared to step through what happens during the consensus algorithm (more specific information will be released on Ed).

Information about how and when to sign up for a check-in will be posted on Ed.

8 Raft Grading

In addition to the test cases, we will be grading the following:

1. 20% of your grade for this project will be focused around the tests you've written.

   Please write exhaustive tests covering a wide range of different failure scenarios. Your README must include a summary of each test you've written and what it does.

2. Test coverage for each of the following files should reach 90%:
   - `node_candidate_state.go`
   - `node_follower_state.go`
   - `node_leader_state.go`

   There is no coverage requirement for other files.

Due to Gradescope time limits, we are enforcing a performance requirement on your tests - each test must complete in **30 seconds individually**. Additionally, your tests must be able to succeed when run with `go test -parallel 8` (which runs tests in parallel).

9 Extra Credit and Capstone Requirements

If you're in a 3-person group or taking CS138 as a capstone, your group must implement one additional feature and demonstrate it works (via tests, a CLI, or benchmarks). Please submit this form if this applies to you.

9.1 Membership Changes

§6 of the Raft paper and §4 of the dissertation

- Configuration options TAs expect:
  - `N` (number of nodes)

- Testcases to consider:
  1. Scaling out (3 → 5 → 7) and scaling down (7 → 3). Adding new nodes should converge on the same most up-to-date leader after the cluster is stabilized to any new cluster size.
  2. Failure case where leader is killed during cluster membership change. Expectation is similar to that of previous: the nodes should cover the most up-to-date leader after the cluster is stabilized.
- Acceptable implementations:
  - Cluster size should change gradually i.e. you must assume some cooling-down time before the cluster leader stabilizes in the new configuration. This cooling-down period should not exceed 2 minutes (provided you do not pause the network in the cluster)
  - You are free (and encouraged) to add new methods to the gRPC protobuf file and generate all pg.go files accordingly. One small caveat: any change you make should be documented and also not delete any existing functionality in the protobuf.
  - You are expected to write at least one case that we can uncomment and run manually for this feature.
  - Suggested implementations (because TAs can answer questions about this):
    1. Add a new client command (e.g. `CONFIGURATION_CHANGE`) that includes an updated configuration + list of `RemoteNodes` which nodes can forward to the leader.
    2. Leader should add a new replicated log entry when informed of the cluster change (add a new type of log entry) + update its internal list of peers and configuration.
    3. Followers, on receiving AppendEntries containing this replicated log entry, should update their internal Peers before storing this log locally.

9.2 Log Compaction

§7 of the Raft paper and §5 of the dissertation

- Configuration options TAs expect:
  - `Threshold` (integer describing the maximum number of logs permissible before compaction is triggered)
- Testcases to consider:
  1. Inspect committed log size before and after the threshold point. After the threshold point, you should see fewer than threshold entries in

the log, with the first entry in the log being: a comma-separated bytestream containing the compressed version of all the previous logs (you can use any compression algorithm you like). You will need to add a new `LOG_TYPE` to your system for this log type.

2. You are free (and encouraged) to add new methods to the gRPC protobuf file and generate all `pg.go` files accordingly. One small caveat: any change you make should be documented and also not delete any existing functionality in the protobuf.

3. You are expected to write at least one test case that we can uncomment and run manually for this feature.

- Acceptable implementations:
  - Modify the leaders and the follower states so that the leader notifies followers they should compact. The leader should also compact its logs of its own volition before doing so.

9.3 Web application to control and interface with a Raft node (using gRPC)

Refer to this repo for an example on `node.js` and `grpc-web`

- Configuration options TAs expect:
  - Use Docker Compose to build and run two services: one Docker container that runs our Golang server, and another Docker container that runs a web server that (a) proxies requests through to the first web server over gRPC, and (b) serves a webpage on a Chrome browser that exposes commands TAs can use (make sure we can choose which port on our local machine we can hit to get this webpage back). The webpage itself should be a Javascript clone of client.go, and offer all the options in that file. Provide a single command that launches everything.
- Testcases to consider:
  1. Full parity with client.go. Every case in that file should be reflected in JS.
  2. You are expected to write at least one test case that we can uncomment and run manually for this feature. Use Selenium or PhantomJS (or some other headless browsing technology) for testing the interface.
- Acceptable implementations:
  - Any implementation that creates a Javascript clone of client.go that uses gRPC to talk to nodes.

9.4 Snapshots with Chandy-Lamport

- Configuration options TAs expect:
  - No strict configurations. Snapshots should be manually triggered, so this feature should not conflict with tests designed solely for non-snapshot implementing versions
  - For extra security, feel free to gate it behind a feature flag (more simply: add a Boolean that disables whether or not someone can trigger a manual snapshot)
- Acceptable implementations:
  - You are free (and encouraged) to add new methods to the gRPC protobuf file and generate all pg.go files accordingly. One small caveat: any change you make should be documented and also not delete any existing functionality in the protobuf.

If you choose to implement one of these, please drop by TA hours or email the TA list to discuss your plan first and get any questions answered.

Feedback

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS138 document by filling out the anonymous feedback form.