Project 3: Raft

Due: 11:59 PM, Apr 10, 2018

Contents

1 Introduction

An important part of creating 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[1] algorithm. 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

[1] `https://en.wikipedia.org/wiki/Paxos_(computer_science)`

practical protocol. In this spirit, a group of researchers at Stanford (Diego Ongaro and John Ousterhout) have developed the Raft protocol[2], 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).

The Raft creators and others have created numerous resources about how the protocol works. As we saw in class, the visualization from The Secret Lives of Data[3] is a great introduction to the protocol. In addition to this, the official Raft website[2] has numerous resources available. We urge you to reference the "In Search of an Understandable Consensus Algorithm (Extended Version)"[4] paper for further details about the protocol. We have taken some bits from the paper and included them in this document for your reference. Lastly, if you would like to learn even more, we suggest you look at Diego Ongaro's dissertation[5], where he discusses in more detail topics such as how clients should interact with a Raft cluster, etc.

Software that makes use of Raft usually works by interpreting the entries in the log as input to a state machine. For this project, we have provided you with code that will calculate the next step of a hash chain[6] each time it finds an ADD command in the log, based on a starting value sent by an INIT command. When you move on to implementing the fourth project, Puddlestore, you will replace the hash chain operations and commands with ones more relevant to implementing a file system.

2 Raft Overview

A Raft cluster typically contains either three or five servers, which can continue to make progress as long as $\left\lfloor \frac{N}{2} \right\rfloor + 1$ nodes remain live. An odd number of servers reduces the chances of split votes in the election phase. Each server is responsible for running the Raft state machine.

The Raft protocol can be broken down into two major components that you will have to implement: Leader Election and Log Replication.

For your reference we have included a summary of the Raft protocol state transition diagram in Figure 1 and a cheatsheet summary of the consensus algorithm in Figure 2.

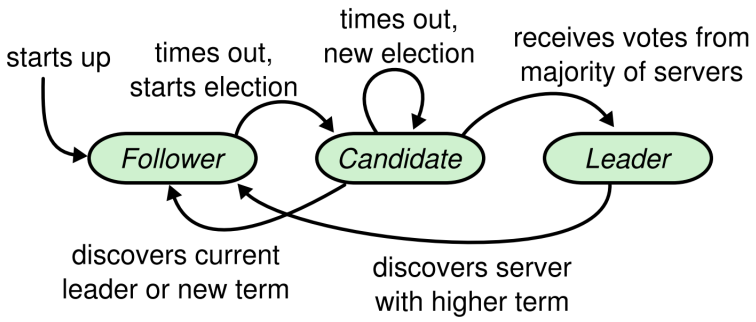

Figure 1: This figure is from the Raft paper, it describes the Raft state transition.

[2] http://raft.github.io/
[3] http://thesecretlivesofdata.com/raft/
[4] https://ramcloud.stanford.edu/raft.pdf
[5] https://github.com/ongardie/dissertation/blob/master/online.pdf?raw=true
[6] https://en.wikipedia.org/wiki/Hash_chain

2.1 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. A node starts out in the FOLLOWER state and if it does not receive a heartbeat within a predefined timeout, it transitions into the CANDIDATE state. Once in the CANDIDATE state, a node votes for itself and sends vote requests to everyone else in the cluster. To a first approximation, if a node receives a vote request before casting a vote for someone else in that term, then they cast a vote for the requester. If the node receives a majority of votes, then it moves to the LEADER state. Once in the LEADER state, the node appends a no-op to the log (§8 of the paper) and sends out heartbeats (AppendEntriesRPCs) to everyone else so that the other nodes know who the new leader is.

The above description of voting for a CANDIDATE is simplified, as there is an extra condition: the correctness of Raft depends on a couple of related mechanisms. A node votes for a candidate if it hasn't voted for anyone else in the election term, and if the log of the candidate is at a higher term, or, if in the same term, at least as complete as the voter's log.

2.2 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. It is based on the AppendEntriesRPC, which is periodically initiated by the leader. The leader accepts requests from clients, adds entries to its log, replicates these entries to a majority of the nodes, commits the entries to the log to allow its followers to feed the entries to their state machines, and then replies to the clients.

Raft leaders are responsible for tracking what log entries have been successfully sent to each node, and don't let the replicated state machines use log entries until they have been propagated to a majority of the nodes. Again, refer to the Raft paper and to Figure 2 for details.

2.3 Client Interaction

Followers, candidates, and a leader form a Raft cluster, which serves Raft clients. A Raft client establishes a session with a Raft cluster by sending a RegisterClientRPC request. The log index of the entry corresponding to this request will be the Client ID returned to the client after the entry has been committed. The client then sends requests to the cluster and gets replies with the results once the corresponding log entries have been committed and fed to the state machine. Replies are cached with the Client ID and a sequence number to avoid executing requests multiple times due to retransmissions.

If the Raft node a client connects to is not the leader node, the node returns a hint of the leader node, and the client follows the hint to reach the leader. This process will be repeated multiple times if the client attempts to connect during an election.

The full implementation of the client has been provided in the stencil code. Refer to the source code for details.

2.4 Other Components

There are other components to building a fully functional Raft cluster, and most of these components have been implemented for you in the support code. For example, we have provided you with a way to use stable storage and code for managing clients that want to interact with the replicated state machine.

3 Implementation

3.1 Raft protocol implementation

For this project, you will be implementing the majority of the Raft protocol. We have provided a lot of the framework to help you concentrate on the core interactions between Raft nodes. You will be implementing:

- Elections (RequestVoteRPC)
- Log replication (AppendEntriesRPC)
- Client Registration
- Client Requests
- gRPC Client and Server Wrappers

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

Furthermore, you'll see a `cmd` directory with two 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. This pattern of storing multiple binaries in a `cmd` directory is a common pattern in Go.

The example client in `raft-client` can control the hash chain implementation in `hashmachine`. The code you must write is marked with `// TODO: Students...` comments and is located in various files in the `raft` directory. Feel free to add whatever support code you need, but try not to change the public APIs too much.

You should implement the following functions:

- `func (r *RaftNode) doFollower() stateFunction`
- `func (r *RaftNode) doCandidate() stateFunction`
- `func (r *RaftNode) doLeader() stateFunction`

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 (§3.5); also used as heartbeat (§3.4).

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 (§3.3)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§3.5)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§3.5)
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 (§3.4).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§3.6)
lastLogTerm	term of candidate's last log entry (§3.6)

Results:

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

Receiver implementation:

1. Reply false if term < currentTerm (§3.3)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§3.4, §3.6)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§3.5)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§3.3)

Followers (§3.4):

- 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 (§3.4):

- 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 RPC (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§3.4)
- If command received from client: append entry to local log, respond after entry applied to state machine (§3.5)
- 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 (§3.5)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§3.5)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§3.5, §3.6).

Figure 2: 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).

Each of these functions 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:

```
func (r *RaftNode) doLeader() state {
    return r.doFollower
}
```

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:

```
for {
    select {
    case msg := <-r.appendEntries:
        msg.reply <- AppendEntriesReply{
            r.GetCurrentTerm(),
            true,
        }
    }
}
```

The full list of channels you should handle are:

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

We've also provided example helper function signatures for you to fill in. Feel free to ignore them if they don't fit into your code, or if you want to change their signatures. They are:

- `func (r *RaftNode) handleAppendEntries(msg AppendEntriesMsg) (resetTimeout, fallback bool)`
- `func (r *RaftNode) requestVotes(electionResults chan bool, currTerm uint64)`
- `func (r *RaftNode) handleCompetingRequestVote(msg RequestVoteMsg) (fallback bool)`
- `func (r *RaftNode) sendHeartbeats() (fallback, sentToMajority bool)`
- `func randomTimeout(minTimeout time.Duration) <-chan time.Time`

In addition to the structure we provide in these functions, we also provide stencil code to deal with persisting Raft state and log entries to disk. This means that you can kill and restart a Raft node, and it'll start up again with the same persistent state (current term, list of other nodes, outstanding client requests, etc.) and log entries as it had before.

Note that 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 as the old node. To avoid this, you can either ensure that new nodes you start have distinct ports, or you can delete the `raftlogs` directory which contains this persisted data.

Finally, you'll need to implement certain gRPC functions as in Tapestry. See the next section for more details.

3.2 gRPC implementation

In Tapestry, you used gRPC, Google's library for RPC, to facilitate communication between nodes over the network. In this project, we also use gRPC, but this time we're asking you to implement more of the RPC code yourself.

In particular, we provide you with a Protocol Buffer (protobuf) file, `raft_rpc.proto`, which defines what RPC calls Raft can deal with, along with what information should be passed into and returned from those calls. We ask you to use this file to generate a Go file that will define all the relevant RPC calls and structs for use in the rest of your Go code.

3.2.1 Protobuf file

To do this, you must first install the Go protobuf compiler by running:

```
$ go get -u github.com/golang/protobuf/protoc-gen-go
$ cd $GOPATH/src/github.com/golang/protobuf && go install ./...
```

You can then compile `raft_rpc.proto` to a file called `raft_rpc.pb.go` by running:

```
$ export PATH=$PATH:$GOPATH/bin
$ /course/cs1380/bin/cs138_protoc -I . ./raft_rpc.proto --go_out=plugins=grpc:.
```

Note that on your own machines you can install and run `protoc` using Homebrew on Mac or Launchpad on Ubuntu. For Windows, we recommend compiling the file on a department machine using the above command and then copying it to your local computer.

It may seem somewhat unnecessary to learn an entirely new syntax to write a file like this just to immediately compile it to Go. You might wonder, why not just write the Go file that defines the RPC interface to begin with?

The main reason one would use Protocol Buffers and gRPC for a system like this is to provide easy cross-platform compatibility. Because a Protocol Buffer defines a standard, language-agnostic syntax, we could easily use it to generate similar RPC implementation files in other languages.

For instance, we could compile Raft's protobuf file into JavaScript, and then with very little work we could use that in a JavaScript application to make requests directly to our Raft nodes! Even though our Raft implementation is in Go, we could use this to easily make a web application that interacts directly with our Raft cluster.

In fact, we will give extra credit to any group that does exactly this: create a web interface for Raft, which, for a given Raft node, displays some information about the node and some interactive functionality to change its behavior. Enabling and disabling communication (via `NetworkPolicy`) could be a fairly straightforward interaction and it would require minimal changes to the RPC interface in `raft_rpc.proto`.

3.2.2 gRPC Client and Server functions

In addition to compiling the protobuf file yourself, we also ask that you implement the client and server sides of each RPC function. Client side functions take in parameter(s), marshall them (if needed) to the required gRPC format, and then sends them over the network to another node, addressed by its IP address and port.

Server side functions are responsible for handling incoming requests from gRPC, passing off the provided data to the relevant local implementation, and then returning the desired output data in the form that gRPC expects.

We've already defined the type signatures for each of these functions, and they can be found in `raft_rpc_client.go` and `raft_rpc_sever.go`, respectively. You are responsible for filling in the functions in both of those files that are marked with `// TODO: Students...`.

We have provided one filled-in function in both files, which you can use as a guide for the rest of the functions. Much of this implementation will be simply error checking and marshalling data to the relevant gRPC format. In these functions, you must also check if connections are allowed between the two nodes trying to communicate. The inline comments in these files and the "Testing" section below have more details.

The reason we ask you to fill in these two sides of the RPC system is to get you comfortable with the details of making and responding to gRPC requests. Though the code you write here is not the most exciting, it is important! Error handling, especially along the network barrier where many error cases can appear, is some of the most important code you will write to maintain a robust, reliable system. This is also good practice for your final project, Puddlestore, where you'll define the entire RPC protocol and associated functions yourself.

4 Testing

We expect to see several good test cases. This is going to be worth a portion of your grade. Exhaustive Go tests are sufficient. You can check your test coverage by using Go's coverage tool[7].

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 under `network_policy.go`, and see how it can be used in `raft-node`. To use it, you first have to implement the RPC client and server functions, and explicitly check if connections should be

[7] `http://blog.golang.org/cover`

allowed between two nodes that are attempting to communicate. Once that's implemented, you can use each RaftNode's `NetworkPolicy` struct in your tests to simulate network splits of different kinds.

Note that the logs of the nodes are stored on disk, which ensures the nodes can resume their states after reboots. You will want to remove the logs if you want to start up new nodes with the same ports, presumably in your unit tests.

4.1 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:

```
$ go get -u ./...
```

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

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

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

- `raft-node`

  This is a CLI that serves as a console for interacting with Raft, creating nodes, and querying state. We have kept the CLI simple but you are welcome to improve it as you see fit.

  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

  You get the following set of commands available to you in the terminal:

  - `debug <on|off>`: turn debug messages on or off.
  - `enable all | enable <send|recv> <addr>`: enables sending or receiving RPCs from an address, or from all addresses.
  - `disable all | enable <send|recv> <addr>`: disables sending or receiving RPCs from an address, or from all addresses.
  - `state`: prints out the local node's view of the cluster's state.
  - `log`: print out the current list of logs.
  - `leave`: gracefully leave the cluster.
  - `exit`: quit the CLI.

- `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 get the following commands:

  - `init <value>`: send an value to initialize the hash machine with.
  - `hash`: instruct the state machine to perform another round of hashing.

You're encouraged, but not required, to write further client applications using the `client` package if your application is in Go, or generated gRPC functions in another language otherwise.

5 Style

CS 1380 does not have an official style guide, but you should reference "Effective Go" for best practices and style for using Go's various language constructs.

Note that naming conventions in Go can be especially important, as using an upper or lower case letter for a method name affects the method's visibility outside of its package.

At a minimum, 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.

6 Getting Started

To get started, you will want to merge in the Raft support code:

```
git remote add stencil https://github.com/brown-csci1380/stencil-s18.git
git pull stencil master
```

Then, perform a global find-and-replace to change import paths from those pointing to `stencil-s18` to your group's repository.

Next, follow the instructions in the gRPC section above to generate a Go file from the protobuf file:

```
$ /course/cs1380/bin/cs138_protoc -I . ./raft_rpc.proto --go_out=plugins=grpc:.
```

Finally, you can follow the instructions from the "Building and Running" section to get going.

7 Extra Credit

If you're in a 3-person group, your group must implement one additional feature and demonstrate it works (via tests, a CLI, or benchmarks). 2-person groups can start thinking about these too, as you may want to implement them for an A-level design of your final project, Puddlestore.

Some ideas for extra credit include:

- Membership Changes (§6 of the Raft paper and §4 of the dissertation)
- Log Compaction (§7 of the Raft paper and §5 of the dissertation)
- Web application to control and interface with a Raft node (using gRPC)
- Snapshots, using the Chandy-Lamport or another algorithm

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.

8 Handing in

You need to write a README documenting your tests, any bugs in your code, any extra features you added, and anything else you think the TAs should know about your project. Once you have completed your README and project, you should hand in your `raft` by running

```
/course/cs1380/bin/cs1380_handin raft
```

to deliver us a copy of your code.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS1380 document by filling out the anonymous feedback form:

`http://cs.brown.edu/courses/cs138/s18/feedback.html`.