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 in the system. The problem of distributed consensus has been around for a long time and has typically been solved using implementations of the popular Paxos algorithm. However Paxos has been shown to be difficult to fully understand, let alone to implement. The difficulties related to Paxos has spawned much work over the years in trying to make it a more practical protocol. In this spirit, a group of researchers at Stanford (Diego Ongaro and John Ousterhout) have developed the Raft protocol, 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 is a great introduction to the protocol.

In addition to this the official Raft website\footnote{https://ramcloud.stanford.edu/raft.pdf} has numerous resources available. We urge you to reference the “In Search of an Understandable Consensus Algorithm (Extended Version)\footnote{https://github.com/ongardie/dissertation/blob/master/online.pdf?raw=true}” 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\footnote{https://en.wikipedia.org/wiki/Hash_chain} 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\footnote{https://en.wikipedia.org/wiki/Hash_chain} each time it finds an ADD command in the log, based on a starting value sent by a 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 $\lfloor \frac{N}{2} \rfloor + 1$ 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.

![Raft State Transition Diagram](image)

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

### 2.1 Leader Election

Leader election consists of a Raft cluster deciding which of the nodes in its cluster should be the leader of a given term. A node starts out in Follower state and if it does not receive a heartbeat within a predefined timeout it transitions into 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 (AppendEntriesRPC’s) to everyone else, so that the other nodes know who the new leader is.

The description of voting for a Candidate above 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 AppendEntries RPC, 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 entry to the log to allow the received followers to feed the entry 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 up a Raft cluster, which serves Raft clients. A client establishes a session with the Raft clusters by sending a request with CLIENT_REGISTRATION operation, and 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 clusters and gets replies with the results once the corresponding log entries have been committed and fed to the state machine.

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, and this process will be tried multiple times in case of in-progress elections. A requests is cached with the Client ID and a request serial number to avoid being executed twice due to retransmissions.

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

2.4 Other

There are other components to building a fully functional Raft cluster, most of these other components your TAs have 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.
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).
3 Implementation

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

- Elections (RequestVote RPC)
- Log replication (AppendEntries RPC)
- Client registration
- Client requests

We have provided you with an example client in client.go that can control the hash chain implementation in machine.go. The code you must write is marked with //TODO: students comments and is located in the raft_states.go file in the raft directory. Feel free to add whatever support you need, but you should implement the following functions:

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

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:

```go
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 AppendEntries RPC:

```go
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 to help hint at what you’ll want to consider. 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) handleCompetingRequestVote(msg RequestVoteMsg) bool`
- `func (r *RaftNode) requestVotes(electionResults chan bool)`
- `func (r *RaftNode) sendHeartBeats() (fallBack, sentToMajority bool)`
- `func (r *RaftNode) makeElectionTimeout() <-chan time.Time`

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](http://blog.golang.org/cover).

To help you test out the behaviour 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 `testing_policy.go`, and see how it can be used in `cli-node.go`.

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

a. `cli-node.go` - This is a Go program that serves as a console for interacting with Raft, creating nodes and querying state on the local node(s). We have kept the CLI simple but you are welcome to improve it as you see fit.

Assuming your CS138 git repository is located at `~/course/cs138`, you can build and run the CLI as follows

```
$ cd ~/course/cs138/raft
$ go build -o cli
$ ./cli
```

You can pass the following arguments to `cli-node.go`:  

[1] Go’s coverage tool
• -p (int): The port to start the server on. By default selects a random port.
• -c "host:port": Address of an existing Raft node to connect to
• -d=true: Enable or disable debug
• -help: Print usage

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

• debug (on|off)
  - Turn debug messages on or off
• recv (addr) (on|off)
  - Control whether or not you receive RPCs from an address
• send (addr) (on|off)
  - Control whether or not you can send RPCs to an address
• disable
  - Stop sending or receiving RPCs
• enable
  - Allow sending or receiving RPCs
• state
  - Print out the local node’s view of the cluster’s state
• exit
  - Quit the CLI

We have also prepared a sample client in cli-client.go, which allows you to interact with
the hash chain state machine we’ve provided. You can build it by running:

$ cd ~/course/cs138/raft
$ go build -o client -tags 'client'
$ ./client

You get the same ‘debug’ and ‘exit’ commands, as well as:

• init (value)
  - Send an initial value for hashing to the replicated state machine
• hash
  - Instruct the state machine to perform another round of hashing

b. You’re encouraged but not required to write client applications, using the ClientRequestRPC
to test your implementation.

5 Style

CS 138 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

Before you get started, please make sure you have read over, understand, and have set up all the common code.

To get started, clone your and your partner’s Github repositories to your CS 138 course directory. Once you have done that, you will want to merge in the Raft support code:

```
git remote add support git@github.com:brown-csci1380/projects-2016.git
git pull support master
```

7 Ideas For Puddlestore

When you move on to implementing Puddlestore, there are some additional features you may want to implement in Raft. Two big ones you may want are support for membership changes (§6 of the Raft paper and §4 of the dissertation) and support for log compaction (§7 of the Raft paper and §5 of the dissertation).

Note that if you do implement any of them, they will not count as extra credit for this project but will be considered as features for a potential A-level Puddlestore design.

8 Handing in

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. Once you have completed your README and project, you should hand in your `raft` by running

```
/course/cs138/bin/cs138_handin raft
```

to deliver us a copy of your code.

[https://github.com/brown-csci1380](https://github.com/brown-csci1380)
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:

http://cs.brown.edu/courses/cs138/s16/feedback.html