Milestones:

1. Start servers, listen to their designated port, and connect to the servers in the configuration file.
2. Implement the client that sends Commands to one server, as read from standard input.
3. Implement the leader election algorithm in Section 5.2 by Ongaro and Ousterhout.
4. Implement the replication algorithm in Section 5.3 by Ongaro and Ousterhout.
5. Iron out any wrinkles that may preclude you from getting it correct.

Server:

States:

6. Candidate
    a. Process calls from the client.
    b. Either buffer commands and process them after the election concludes OR drop the messages
7. Follower
    a. Process calls from the client.
8. Leader
    a. Process calls from the client.
9. Failed
    a. Receive all messages.
    b. Do not respond.
    c. Do not send heartbeat messages.
    d. Do not initiate elections.
    e. Forward command names from clients to the Leader.

Logging:

1. Replicated state machine of each server only logs all the transactions that the system agrees on.
2. Write the command name to a log file ***(server-host-port.log)***

    term, index, command
    term, index, command

Delivery Consistency:

1. Raft does not guarantee messages are delivered in order.
2. Keep track of the response messages that server is expecting.
    a. Especially during leader election, new calls may announce a leader before server receives all RequestVote response messages.

Client:

Purpose:

1. Generate command names to the server.
    a. Uninterpreted strings
        i. Non-empty
        ii. Lower- and upper-case letters and digits
        iii. Dashes and underscores
        iv. NO punctuations and whitespaces
    b. i.e. *insert, select, update*
    c. *For debugging,* use unique strings.
        i. Encode client identities and sequence numbers, and track what command names are processed.
    d. Server will store the names in their log file.
2. Need not wait for acknowledgements that their commands are committed

Guarantee:

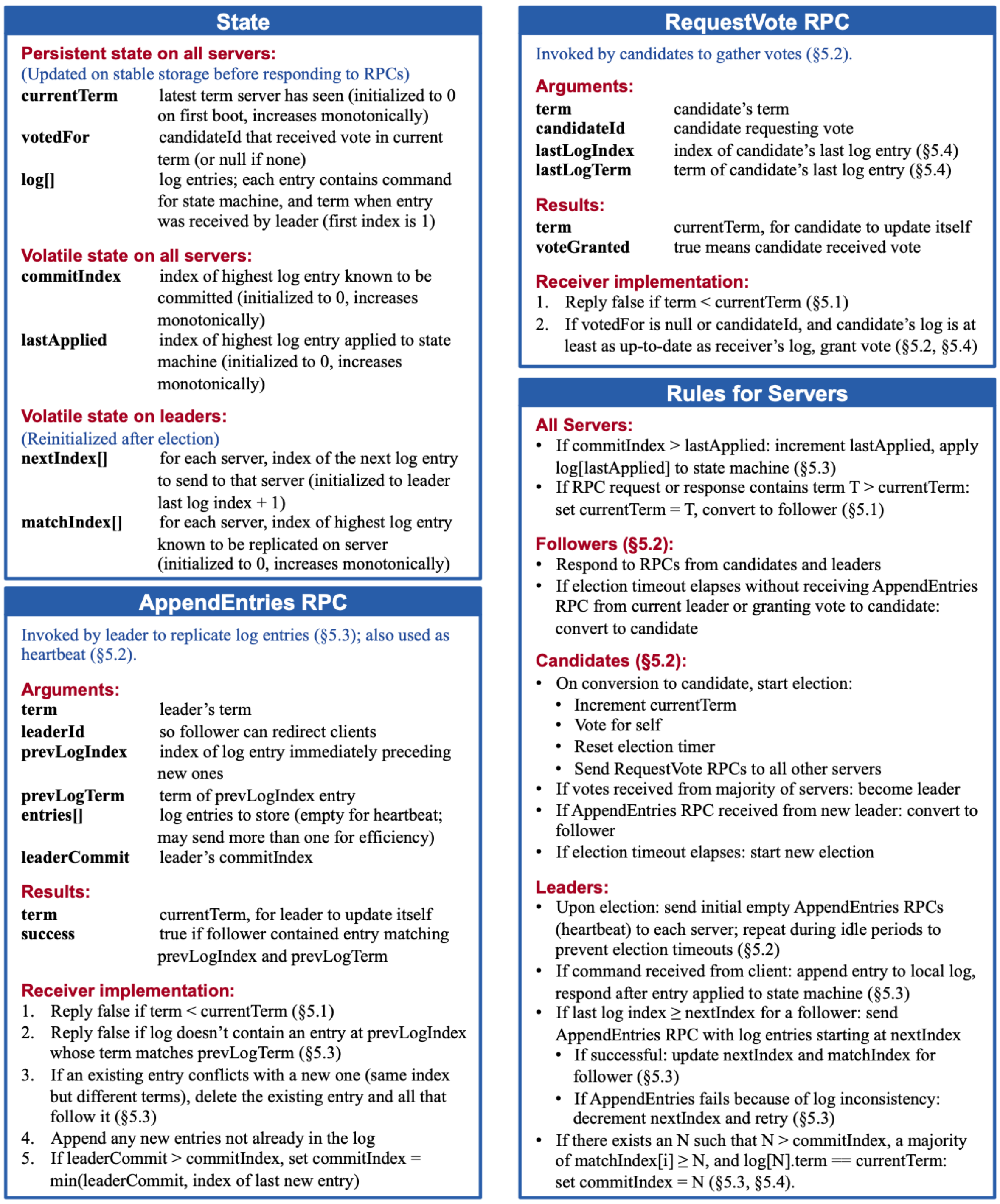
1. State machines commit the commands in the same order.
2. Does not need to commit all commands submitted by clients (most of it).

Communication:

Requirements:

1. Use UDP
2. Google protobuf format
    a. *https://developers.google.com/protocol-buffers/docs/proto3*
    b. *https://developers.google.com/protocol-buffers/docs/gotutorial*
    c. *https://developers.google.com/protocol-buffers/docs/reference/go-generated*
    d. *https://pkg.go.dev/google.golang.org/protobuf/proto*
    e. *https://developers.google.com/protocol-buffers/docs/reference/go/*
3. Server
    a. Listen to port designated on setup.
    b. Use full *server-host:server-port* to identify itself to other servers.
    c. Make sure all UDP packets from this address are sent to other servers.
    d. *Net.ReadFromUDP* function returns sender address to the receiver.
    e. Receiver need to send a response back??? (TO CLARIFY)

4. Message specification (see Fig 2 below)



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 [28] describes the algorithm more precisely.

Listing 1 is the given raft message format for go:

```
message LogEntry {
        uint64 Index=1;
        uint64 Term=2;
        string CommandName=3; // Empty string is noop
}

// AppendEntries RPC
message AppendEntriesRequest {
        uint64 Term=1;
        uint64 PrevLogIndex=2;
        uint64 PrevLogTerm=3;
        uint64 LeaderCommit=4;
        string LeaderId=5;
        repeated LogEntry Entries=6;
}

message AppendEntriesResponse {
        uint64 Term=1;
        bool   Success=4;
}

// RequestVote RPC
message RequestVoteRequest {
        uint64 Term=1;
        uint64 LastLogIndex=2;
        uint64 LastLogTerm=3;
        string CandidateName=4;
}

message RequestVoteResponse {
        uint64 Term=1;
        bool VoteGranted=2;
}

// Wrapper
message Raft {
        oneof Message {
              AppendEntriesRequest AppendEntriesRequest=1;
              AppendEntriesResponse AppendEntriesResponse=2;
              RequestVoteRequest RequestVoteRequest=3;
              RequestVoteResponse RequestVoteResponse=4;
              string CommandName=5;
        }
}
```

Listing 1: Raft message format

Interactions:

Server and client run as **foreground processes**.

Server Process:

Command Line: go run raftserver.go server-host:server-port filename

1. Arguments to start the server.
    a. Own identity
        i. *server-host:server-port*
        ii. Listens on its port
    b. File of all server identities
        i. Format is as follows...

            localhost:2000
            127.0.0.1:2020
            server2:2000

        ii. Server must ensure that its identity is in the file.
2. Commands
    a. log
        i. Print the log entries.
    b. **print**
        i. Print the current term, leader voted for, state, commitIndex, and lastApplied index, nextIndex, and matchIndex.
    c. **resume**
        i. Switch a server into the normal execution states.
    d. **suspend**
        i. Switch a server to a **suspended state**.
        ii. **Will** receive messages from the client and other servers.
        iii. **Will not** respond to any message while in this state.
        iv. **Will not** send any heartbeat messages.

Client Process:

Command Line: *go run raftclient.go server-host:server-port*

1. Read a line from its standard input stream.
2. Send string as a ***CommandName*** of a **message Raft** (class, refer to Listing 1) to the server. (unless it is "exit")
3. **exit:** allows a client node to exit the system.
    a. READ from string.

Command Line Arguments:

Store all code into directory called “group-n” (n is our group number)

Server:

```
go run raftserver.go server-host:server-port filename
```

Client

```
go run raftclient.go server-host:server-port
```

Should be able to run on localhost and different host.

Verification:

1. Start a network of three to ten servers.
2. Clients connect to random servers and issue arbitrary commands.
    a. **Commands CAN be unique.**
3. Server write committed commands to their log file.
4. Compare contents of log file.
    a. Correct, if same sequence of command.