Raft Distributed Consensus

AUTHORS

Kwok Keith
Toh Hong Jing

March 25, 2024

https://github.com/kwokkeith/RaftConsensus

Contents

1 Protocol Explanation **1**
1.1 Leader Election Algorithm 1
1.2 Replication Algorithm 8

2 Request and Response Implementation **15**
2.1 Networking Code 15
2.2 Election Messages 18
2.3 Log Messages 19

3 Heartbeat Mechanism **20**
3.1 Single Heartbeat 20
3.2 Periodic Heartbeat 21

4 Timeouts **22**

5 Progress and Deadlock Freedom **22**

List of Figures **I**

Nomenclature **II**

References **III**

6 Appendix A **IV**

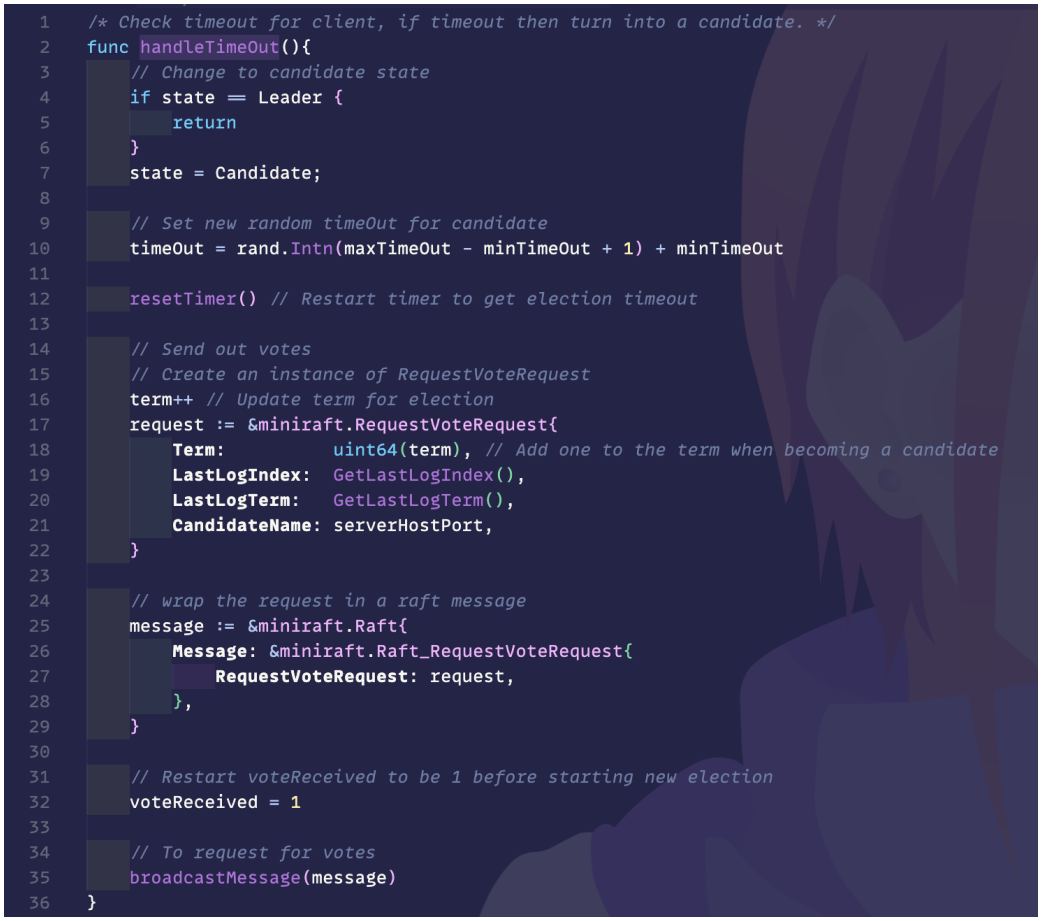
1 Protocol Explanation

This section traces each function of the protocol implemented to the protocol desciption of Ongaro and Ousterhout (2014). Two algorithms from Onagaro and Ousterhout, specifically the leader election algorithm (section 5.2 of Ongaro and Ousterhout) and the replication algorithm (section 5.3 of Ongaro and Ousterhout) was implemented and would be explained below.

1.1 Leader Election Algorithm

```go
// To set timer for handling timeout of servers
func resetTimer() {
    timerMutex.Lock()
    defer timerMutex.Unlock()

    if timerIsActive {
        // Stop the current timer. If Stop returns false, the timer has already fired.
        if !timeOutCounter.Stop() {
            // If the timer already expired and the handleTimeOut function might be in queue,
            // try to drain the channel to prevent handleTimeOut from executing if it hasn't yet.
            select {
            case <-timeOutCounter.C:
            default:
            }
        }
    }

    // Regardless of the previous timer's state, start a new timer.
    timerIsActive = true
    timeOutCounter = time.AfterFunc(time.Second*time.Duration(timeOut), func() {
        handleTimeOut()
        // After the timer executes, reset timerIsActive to false.
        timerMutex.Lock()
        timerIsActive = false
        timerMutex.Unlock()
    })
}
```

Figure 1: resetTimer starts/restarts the timer for servers in the follower and candidate state



```
1  /* Check timeout for client, if timeout then turn into a candidate. */
2  func handleTimeOut(){
3      // Change to candidate state
4      if state == Leader {
5          return
6      }
7      state = Candidate;
8
9      // Set new random timeOut for candidate
10     timeOut = rand.Intn(maxTimeOut - minTimeOut + 1) + minTimeOut
11
12     resetTimer() // Restart timer to get election timeout
13
14     // Send out votes
15     // Create an instance of RequestVoteRequest
16     term++ // Update term for election
17     request := &miniraft.RequestVoteRequest{
18         Term:          uint64(term), // Add one to the term when becoming a candidate
19         LastLogIndex:  GetLastLogIndex(),
20         LastLogTerm:   GetLastLogTerm(),
21         CandidateName: serverHostPort,
22     }
23
24     // wrap the request in a raft message
25     message := &miniraft.Raft{
26         Message: &miniraft.Raft_RequestVoteRequest{
27             RequestVoteRequest: request,
28         },
29     }
30
31     // Restart voteReceived to be 1 before starting new election
32     voteReceived = 1
33
34     // To request for votes
35     broadcastMessage(message)
36 }
```

Figure 2: handleTimeOut checks for timeout of a client and if so turns the client into a candidate to run for leader election

The handleTimeOut function is triggered when a Follower server times out according to the timeOutCounter as started from the resetTimer function (see Figure 1).

Refer to Figure 2, when the server times out, the Follower server would change into the Candidate state (see line 7). It would randomize its election timeout (see line 10) before starting the election timer (see line 12) to reduce the likelihood of another split vote in the new election. It next increments its current term (see line 16) and sends a RequestVote Remote Procedural Call (RPC) to all other servers as seen in lines 17 to 35. The candidate would vote for itself and hence the voteReceived is instantiated with an initial value of 1 (see line 32). Note that the election timer would restart the election process if it times out, this represents a failed election and assumes that there exist a split vote.

```
1  // If receiving a request vote
2  func handleRequestVoteRequest(message miniraft.Raft_RequestVoteRequest){
3      // Check if server has the pre-requisite to be voted
4      // Choose candidate with log most likely to contain all committed entries
5      // This is achieved by comparing logs
6      var vote miniraft.RequestVoteResponse;
7
8      if (message.RequestVoteRequest.LastLogTerm < GetLastLogTerm() ||
9      (message.RequestVoteRequest.LastLogTerm == GetLastLogTerm() &&
10     message.RequestVoteRequest.LastLogIndex < GetLastLogIndex() )){
11         // Do not give vote to candidate since he is out-dated
12         vote = miniraft.RequestVoteResponse{
13             Term:          uint64(term), // To update candidate
14             VoteGranted:   false,
15         }
16     } else {
17         // Send a vote to the candidate
18         leaderVotedFor = message.RequestVoteRequest.CandidateName
19
20         vote = miniraft.RequestVoteResponse{
21             Term:          message.RequestVoteRequest.GetTerm(), // Take candidate's ter
22             VoteGranted:   true,
23         }
24
25         // Update own term
26         term = int(message.RequestVoteRequest.GetTerm())
27     }
28
29     // Set new random timeout
30     timeOut = rand.Intn(maxTimeOut - minTimeOut + 1) + minTimeOut
31
32     // Create Message wrapper for raft message
33     voteResponse := &miniraft.Raft{
34         Message: &miniraft.Raft_RequestVoteResponse{
35             RequestVoteResponse: &vote,
36         },
37     }
38
39     // Send message to candidate
40     SendMiniRaftMessage(message.RequestVoteRequest.GetCandidateName(), voteResponse)
41 }
```

Figure 3: handleRequestVoteRequest() handles the procedures to respond to a vote request from a candidate server

Refer to Figure 3, this function handles the response to a vote request from a candidate. The follower server would choose the candidate with the log most likely to contain all committed entries by comparing its logs with the candidate's log. If the voting server's log is more complete it denies the vote. That is, the term of its last entry is more than the term of the last entry of the candidate or if they are equal, then the index of its last entry is more than the index of the candidate's (see lines 8 to 15).

Otherwise, the voting server would vote for the candidate and sends a vote granted RequestVoteResponse message (see lines 17 to 40).

```go
func handleRequestVoteResponse(message miniraft.Raft_RequestVoteResponse){
    // Check if state is still candidate
    updateServersKnowledge()
    if state == Candidate {
        // Count up if vote received is granted
        if message.RequestVoteResponse.VoteGranted {
            voteReceived++
            if voteReceived > len(servers) / 2 {
                // Received majority
                setupLeader()
            }
        }
    }
}

```

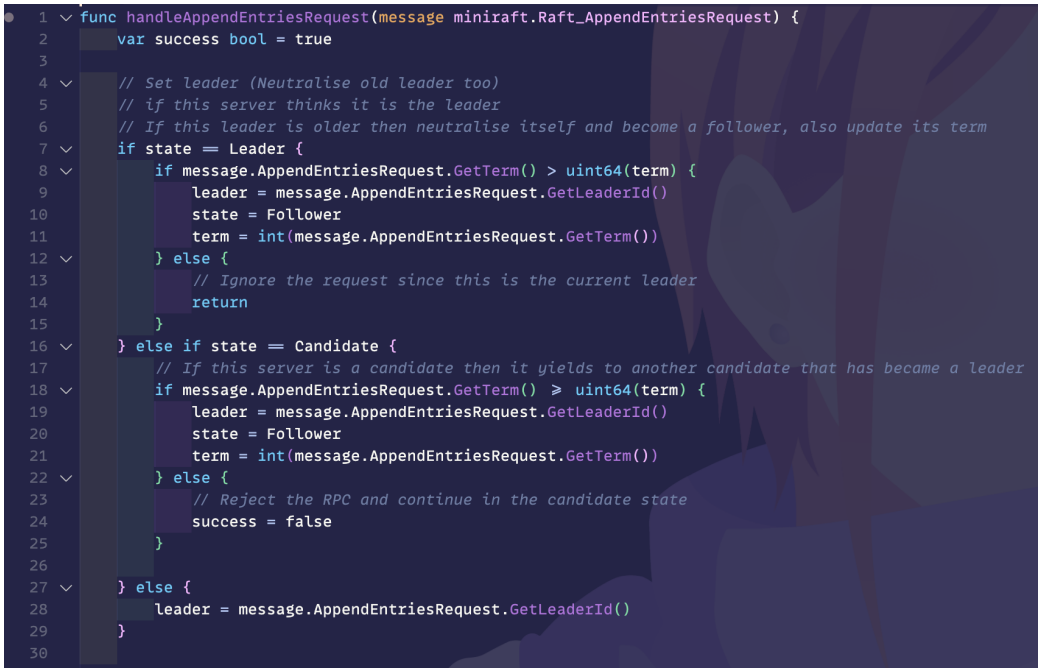
Figure 4: handleRequestVoteResponse() handles the procedures to detect the result of the election

Refer to Figure 4, the candidate now checks if it has received a vote granted from majority of the servers. It first updates its knowledge of the servers present in the network. If the vote is granted, the candidate would increment its voteReceived counter. Once a majority of the vote has been received, the candidate would become the leader and executes setupLeader.

```go
/* Function to initialise and setup a leader */
func setupLeader(){
    // Send heartbeat to tell servers, this is the new leader
    SendHeartBeat();
    state = Leader; // Become the new leader
    go SetHeartBeatRoutine(); // Routinely send heartbeat
    leader = serverHostPort // Update who the current leader is
    voteReceived = 1 // Reset counter (1 because leader votes for himself)

    // Initialise the nextIndex and matchIndex
    for _, server := range servers {
        nextIndex[server] = int(GetLastLogIndex()) + 1
        matchIndex[server] = 0
    }

    // Stop timer too for timeout
    timerMutex.Lock()
    defer timerMutex.Unlock()
    if timerIsActive {
        // Stop the current timer. If Stop returns false, the timer has already fired.
        if !timeOutCounter.Stop() {
            // If the timer already expired and the handleTimeOut function might be in queue,
            // try to drain the channel to prevent handleTimeOut from executing if it hasn't yet.
            select {
            case <-timeOutCounter.C:
            default:
            }
        }
    }
}
```

Figure 5: setupLeader() initialises and setups the conversion from candidate to leader state

Refer to Figure 5, setupLeader is a function call that handles the procedure to convert a server from a candidate state into a leader state. This function is only performed if the candidate server obtains a majority vote granted message and has not received any heartbeat from another leader. The function proceeds to prioritise sending a heartbeat to all other servers in the network to update them of a leader conversion (see line 4). The candidate server then converts into a leader (see line 5). The leader server now sets up a heartbeat routine to send heartbeats periodically (see line 6). It proceeds to perform the necessary leader setups as stipulated in Figure 2 in Ongaro and Ousterhout's report (refer to Figure 22). The leader server would finally stop the election timeout counter (see lines 17 to 29).



```
1  func handleAppendEntriesRequest(message miniraft.Raft_AppendEntriesRequest) {
2      var success bool = true
3
4      // Set leader (Neutralise old leader too)
5      // if this server thinks it is the leader
6      // If this leader is older then neutralise itself and become a follower, also update its term
7      if state == Leader {
8          if message.AppendEntriesRequest.GetTerm() > uint64(term) {
9              leader = message.AppendEntriesRequest.GetLeaderId()
10             state = Follower
11             term = int(message.AppendEntriesRequest.GetTerm())
12         } else {
13             // Ignore the request since this is the current leader
14             return
15         }
16     } else if state == Candidate {
17         // If this server is a candidate then it yields to another candidate that has became a leader
18         if message.AppendEntriesRequest.GetTerm() >= uint64(term) {
19             leader = message.AppendEntriesRequest.GetLeaderId()
20             state = Follower
21             term = int(message.AppendEntriesRequest.GetTerm())
22         } else {
23             // Reject the RPC and continue in the candidate state
24             success = false
25         }
26
27     } else {
28         leader = message.AppendEntriesRequest.GetLeaderId()
29     }
30
```

Figure 6: If the candidate receives an AppendEntriesRequest/heartbeat message, it would convert into a Follower

Refer to Figure 6, a candidate who receives a heartbeat message, an empty AppendEntriesRequest, would convert itself into a Follower if this request has a term that is not older than its current term. This prevents an old leader which have just revived to interrupt an ongoing election process and only allows a legitimate leader to stop an election. If an illegitimate leader has sent an RPC to the candidate, it would simply reject this RPC. If a candidate converts into a follower, it stops its election algorithm as seen from line 4 in Figure 4.

```go
/* This function sets the heartbeat routine for
leader to send heartbeat in a fixed interval */
func SetHeartBeatRoutine(){
    for state == Leader {
        heartbeatTimerMutex.Lock()
        if !heartbeatTimerIsActive {
            heartbeatTimerIsActive = true
            timeOutCounter = time.AfterFunc(time.Second*time.Duration(heartbeatInterval), SendHeartBeat);
        }
        heartbeatTimerMutex.Unlock()
    }

    timeOutCounter.Stop()
}

```

Figure 7: SetHeartBeatRoutine sets up a timer to send out heartbeat periodically

```go
/* This function is used for the leader to send heartbeats */
func SendHeartBeat(){
    request := miniraft.AppendEntriesRequest{
        Term: uint64(term),
        LeaderId: serverHostPort,
        PrevLogIndex: GetLastLogIndex(),
        PrevLogTerm: GetLastLogTerm(),
        Entries: nil,
        LeaderCommit: uint64(commitIndex),
    }

    // wrap the request in a raft message(AppendEntriesRequest)
    message := &miniraft.Raft{
        Message: &miniraft.Raft_AppendEntriesRequest{
            AppendEntriesRequest: &request,
        },
    }
    broadcastMessage(message)

    heartbeatTimerMutex.Lock()
    heartbeatTimerIsActive = false // Set timer off
    heartbeatTimerMutex.Unlock()
}
```

Figure 8: SendHeartBeat sends out an empty AppendEntriesRequest message to all servers in the network

The heartbeat protocol is handled by two functions, SetHeartBeatRoutine and SendHeartBeat. SendHeartBeat allows a leader to broadcast an empty AppendEntriesRequest message to all servers in the network (refer to Figure 8).

SetHeartBeatRoutine establishes a counter in which a server would send out a heartbeat in a fixed interval as defined by heartbeatInterval global variable (refer to Figure 7). The heartbeatTimerIsActive

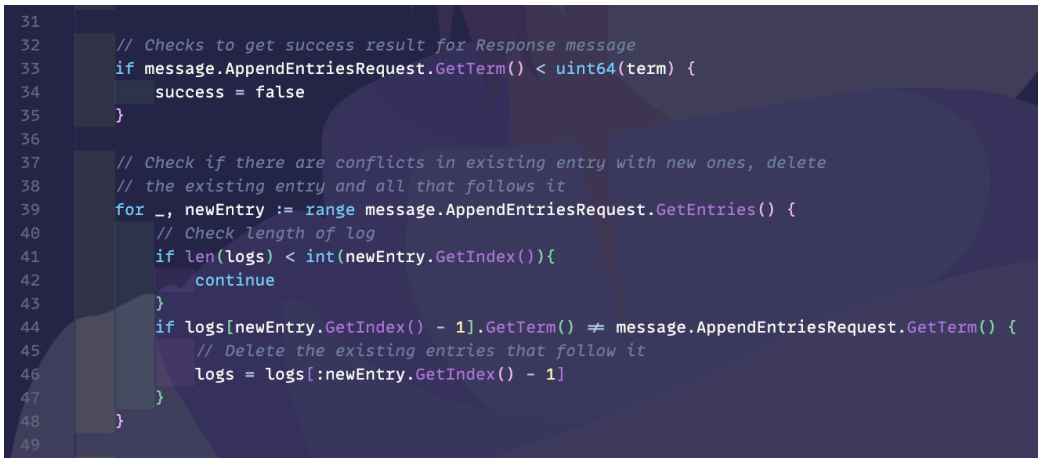
variable would indicate for this counter to be reset when a heartbeat has been sent (see line 21 in Figure 8).

1.2 Replication Algorithm

```go
func handleCommandName(message miniraft.Raft_CommandName) {
    // Check if current leader is this server
    if leader == "" {
        // if leader has not been established drop the message (Could also buffer it)
        return
    } else if leader == serverHostPort {
        // Send out the command name to the others
        // Create a log object
        // TODO: Create log object
        logEntry := miniraft.LogEntry{
            Index: uint64(len(logs)) + 1,
            Term: uint64(term),
            CommandName: message.CommandName,
        }

        // Get last log index and term before sending out AppendEntriesRequest
        var prevLogIndex = GetLastLogIndex()
        var prevLogTerm = GetLastLogTerm()

        logs = append(logs, logEntry) // Leader adds command to its log

        // Leader sends AppendEntries to  message to followers
        request := miniraft.AppendEntriesRequest{
            Term: uint64(term),
            LeaderId: serverHostPort,
            PrevLogIndex: prevLogIndex,
            PrevLogTerm: prevLogTerm,
            Entries: []*miniraft.LogEntry{
                &logEntry,
            },
            LeaderCommit: uint64(commitIndex),
        }

        message := &miniraft.Raft{
            Message: &miniraft.Raft_AppendEntriesRequest{
                AppendEntriesRequest: &request,
            },
        }

        broadcastMessage(message)
    } else {
        // wrap the command in a raft message
        message := &miniraft.Raft{
            Message: &miniraft.Raft_CommandName{
                CommandName: message.CommandName,
            },
        }
        SendMiniRaftMessage(leader, message)
    }
}
```

Figure 9: handleCommandName handles an incoming message from a client

Refer to Figure 9, when a leader has been elected, it begins servicing client requests (see lines 7 to 40). Upon receiving a client request, a leader would create a new log entry with the command received and append it to its log (see lines 10 to 20). The leader then issues AppendEntriesRequest RPCs to each of the other servers to replicate the entry (see lines 23 to 40).



```
31
32      // Checks to get success result for Response message
33      if message.AppendEntriesRequest.GetTerm() < uint64(term) {
34          success = false
35      }
36
37      // Check if there are conflicts in existing entry with new ones, delete
38      // the existing entry and all that follows it
39      for _, newEntry := range message.AppendEntriesRequest.GetEntries() {
40          // Check length of log
41          if len(logs) < int(newEntry.GetIndex()){
42              continue
43          }
44          if logs[newEntry.GetIndex() - 1].GetTerm() != message.AppendEntriesRequest.GetTerm() {
45              // Delete the existing entries that follow it
46              logs = logs[:newEntry.GetIndex() - 1]
47          }
48      }
49
```

Figure 10: Snippet of HandleAppendEntriesRequest, illustrated to explain a client's response to receiving an AppendEntriesRequest message

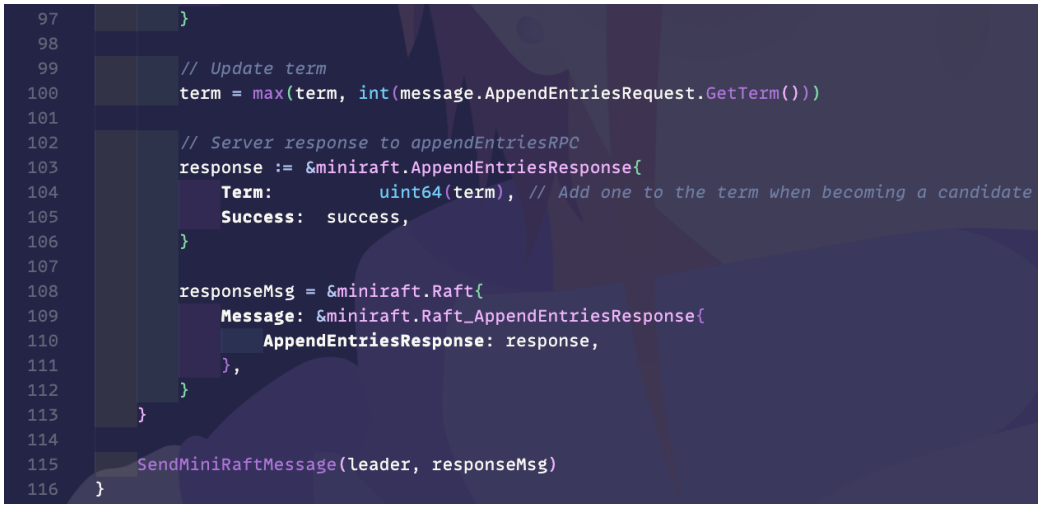
The specifications for the receiver implementation as stipulated by Ongaro and Ousterhout (2014)'s report would be explained below.

Refer to Figure 10, a receiver would reply false, represented by success being false, if the term of the AppendEntriesRequest message is smaller than its current term (see lines 33 to 35). If an existing entry conflicts with a new one, that is, it has the same index but different terms, the server would delete the existing entry and all that follows it (see lines 39 to 48).

```
49
50      responseMsg := &miniraft.Raft{};
51      // Check if previous term, index is the same as the leader, if not then reject
52      if GetLastLogIndex() != message.AppendEntriesRequest.PrevLogIndex ||
53      GetLastLogTerm() != message.AppendEntriesRequest.PrevLogTerm ||
54      len(logs) < int(message.AppendEntriesRequest.GetPrevLogIndex()) {
55          // Reject the appendEntriesRequest
56          response := &miniraft.AppendEntriesResponse{
57              Term: uint64(term),
58              Success: false,
59          }
60
61          responseMsg = &miniraft.Raft{
62              Message: &miniraft.Raft_AppendEntriesResponse{
63                  AppendEntriesResponse: response,
64              },
65          }
66      } else {
67          // Append any new entries not already in the log
68          for _, newEntry := range message.AppendEntriesRequest.GetEntries() {
69              if len(logs) < int(newEntry.GetIndex()){
70                  logs = append(logs, *newEntry)
71              }
72          }
73
74          // Update commitIndex
75          if message.AppendEntriesRequest.GetLeaderCommit() > uint64(commitIndex) {
76              var lastNewEntryIndex uint64 = GetLastLogIndex()
77
78              if message.AppendEntriesRequest.GetLeaderCommit() < lastNewEntryIndex {
79                  commitIndex = int(message.AppendEntriesRequest.GetLeaderCommit())
80              } else {
81                  commitIndex = int(lastNewEntryIndex)
82              }
83          }
84
85          // Write to log file to update server's commited commands.
86          // Update logFile only if commitIndex have changed
87          go UpdateLogFile()
88
```

Figure 11: Snippet of HandleAppendEntriesRequest, illustrated to explain a client's response to receiving an AppendEntriesRequest message

Refer to Figure 11, the receiving server would also reply false if the log does not contain an entry at prevLogIndex whose term matches prevLogTerm of the AppendEntriesRequest message (see lines 52 to 65). If all conditions are above are met, the receiver would then accept this AppendEntriesRequest and append the new entries that does not already exist in its log (see lines 68 to 72). The receiver proceeds to update its commitIndex by setting it to the minimum between the leaderCommit from the AppendEntriesRequest message and the index of its last new entry (see lines 75 to 87). The log file, an external file, is only updated when there is an updated value to its committed index.



```
97          }
98
99          // Update term
100         term = max(term, int(message.AppendEntriesRequest.GetTerm()))
101
102         // Server response to appendEntriesRPC
103         response := &miniraft.AppendEntriesResponse{
104             Term:          uint64(term), // Add one to the term when becoming a candidate
105             Success:  success,
106         }
107
108         responseMsg = &miniraft.Raft{
109             Message: &miniraft.Raft_AppendEntriesResponse{
110                 AppendEntriesResponse: response,
111             },
112         }
113     }
114
115     SendMiniRaftMessage(leader, responseMsg)
116 }
```

Figure 12: Snippet of HandleAppendEntriesRequest, illustrated to explain a client's response to receiving an AppendEntriesRequest message

Refer to Figure 12, the receiving server would update its term (see line 100), if necessary, and it would finally send out a successful AppendEntriesResponse to the leader of the network.

```go
func handleAppendEntriesResponse(message miniraft.Raft_AppendEntriesResponse, senderAddress string) {
    // Check if response is more than the leader's term, if so
    // Leader to convert to follower and updates its term and set leader to be unknown
    if term < int(message.AppendEntriesResponse.GetTerm()) {
        // Convert to follower
        state = Follower
        term = int(message.AppendEntriesResponse.GetTerm())
        leader = ""
        return
    }

    // Check if message has been rejected
    if !message.AppendEntriesResponse.Success {
        // Decrement the known nextIndex for the sender that rejected the message
        if nextIndex[senderAddress] > 1 {
            nextIndex[senderAddress]--;
        }

        // Resend the appendEntriesRequest to this server with decremented index
        var prevLogIndex = nextIndex[senderAddress] - 1

        var indexToSend = prevLogIndex + 1

        // Check if index to send is larger than the size of current log,
        // if larger then we are unable to send a mesasge at index and hence ignore.
        if indexToSend > len(logs){
            return
        }

        // get the previous log, but need to check if there is only 1 message in the log.
        // We need to handle this edge case by setting the previous log term to be 0
        // if there is only 1 message.
        var prevLogTerm uint64 = 0;
        if indexToSend > 1 {
            prevLogTerm = logs[indexToSend-1-1].GetTerm()
        }

        // Prepare entry to resend
        request := miniraft.AppendEntriesRequest{
            Term: uint64(term),
            LeaderId: serverHostPort,
            PrevLogIndex: uint64(prevLogIndex),
            PrevLogTerm: prevLogTerm,
            Entries: []*miniraft.LogEntry{
                &logs[indexToSend-1],
            },
            LeaderCommit: uint64(commitIndex),
        }

        // wrap the request in a raft message(AppendEntriesRequest)
        message := &miniraft.Raft{
            Message: &miniraft.Raft_AppendEntriesRequest{
                AppendEntriesRequest: &request,
            },
        }

        // Send request
        SendMiniRaftMessage(senderAddress, message)
    } else {
```

Figure 13: Snippet of HandleAppendEntriesResponse, illustrated to explain a leader's response to receiving an AppendEntriesResponse message

Refer to Figure 13, a leader upon receiving a response for an AppendEntriesRequest message, would first check if its term is smaller than the term given in the response message. This allows the

network to neutralize old leaders by reverting the old leader, since its term is smaller, into a follower. This server would then act like a follower to the new leader upon receiving the next request from the leader (see lines 4 to 10).

Otherwise, the server would handle the response based on whether it was a successful or unsuccessful one. If the response was unsuccessful, the leader would decrement the next index to send for that server, and tries again to send an AppendEntriesRequest based on this new next index value (see lines 13 to 58).

```
} else {
    // Update index of highest log entry known to be replicated by the follower
    matchIndex[senderAddress] = nextIndex[senderAddress] - 1

    // Increment the follower's next index to be sent if it is not already the last entry in the leader
    if nextIndex[senderAddress] <= len(logs) + 1{
        nextIndex[senderAddress]++;
    }

    // Update commitIndex of leader
    for commitIndex <= len(logs){
        // Check for a majority value
        count := 0
        for _, value := range matchIndex {
            if value >= commitIndex {
                count++
            }
        }

        if count + 1 > len(servers) / 2 {
            commitIndex++
            go UpdateLogFile()
        } else {
            break
        }
    }

    if (commitIndex > 0){
        commitIndex--
    }

    // If nextIndex[senderAddress] is now equal to that of the server's then do
    // not send request again, else send next request
    if nextIndex[senderAddress] == len(logs) + 1 {
        return
    }

    // In case we are at the beginning
    var newPrevLogIndex = 0;
    if nextIndex[senderAddress] > 0 {
        newPrevLogIndex = nextIndex[senderAddress] - 1
    }

    // Else send the next request
    // Prepare entry to resend
    request := miniraft.AppendEntriesRequest{
        Term: uint64(term),
        LeaderId: serverHostPort,
        PrevLogIndex: uint64(newPrevLogIndex),
        PrevLogTerm: (logs[nextIndex[senderAddress] - 1 - 1]).GetTerm(),
        Entries: []*miniraft.LogEntry{
            &logs[nextIndex[senderAddress] - 1],
        },
        LeaderCommit: uint64(commitIndex),
    }

    // wrap the request in a raft message(AppendEntriesRequest)
    message := &miniraft.Raft{
        Message: &miniraft.Raft_AppendEntriesRequest{
            AppendEntriesRequest: &request,
        },
    }

    // Send message to response server
    SendMiniRaftMessage(senderAddress, message)
}
```

Figure 14: Snippet of HandleAppendEntriesResponse, illustrated to explain a leader's response to receiving an AppendEntriesResponse message

Refer to Figure 14, if the response was successful then the leader updates the highest log entry known to be replicated on that server (see line 61). It increments the next index to be sent for that server if it is not already the last entry in the log (see lines 64 to 65). The leader would then update the commitIndex based on the known last replicated log in the other servers (see lines 69 to 88). If the follower has fully replicated its log to be consistent with that of the leader, the leader would not send any other request to the follower (see lines 92 to 94). Otherwise, the leader would send an AppendEntriesRequest for the proceeding entry in its log to repair follower's logs (see lines 97 to 123).

2 Request and Response Implementation

This section explains how the implementation proposed tracks requests and responses to ensure that a response eventually answers all requests.

2.1 Networking Code

```
1   // Function to start server on the defined address
2   func startServer(serverHostPort string){
3       addr, err := net.ResolveUDPAddr("udp", serverHostPort)
4       if err != nil {
5           log.Fatal(err)
6       }
7       serverAddr = addr
8
9       // Generate random timeout
10      timeOut = rand.Intn(maxTimeOut - minTimeOut + 1) + minTimeOut
11
12      // Start communication (looped to keep listening until exit)
13      handleCommunication()
14  }
```

Figure 15: startServer is executed once when the server starts up

The communication happens fully in User Datagram Protocol (UDP). When a server starts up, it sets its address into a global variable (refer to Figure 15). It then calls handleCommunication to listen to its assigned address.

```go
/* This function sets up the listening port for the server to receive
messages from other servers */
func handleCommunication(){
    lstnr, err := net.ListenUDP("udp", serverAddr)
    listener = lstnr
    if err != nil {
        log.Fatal(err)
    }

    // Start Communication
    for {
        data := make([]byte, 65536)

        // Stop any timer that was running
        // Start a timer for time to check timeout if current server is not a leader
        if state != Leader {
            resetTimer()
        }

        length, addr, err := listener.ReadFromUDP(data)
        if err != nil {
            log.Fatal(err)
        }

        // Debug message to check message received
        // log.Printf("From %s: %v\n", addr.String(), data[:length])

        // Handles the message based on its type
        go handleMessage(data[:length], addr.String());
    }
}
```

Figure 16: handleCommunication starts the server listening to its assigned address

```go
/* Function to handle the message based on its type */
func handleMessage(data []byte, senderAddress string){
    // Unmarshal the message
    message := &miniraft.Raft{}
    err := proto.Unmarshal(data, message)
    if err != nil {
        log.Printf("Failed to unmarshal message: %v\n", err)
    }

    // Switch between different types of protobuf message
    mutex.Lock()
    defer mutex.Unlock() // Release the mutex
    if suspended {
        fmt.Println("Received message but suspended.")

        return
    }

    switch msg := message.Message.(type) {
    case *miniraft.Raft_CommandName:
        fmt.Println("Received CommandName response: ", msg.CommandName)
        handleCommandName(*msg)
    case *miniraft.Raft_AppendEntriesRequest:
        fmt.Printf("Received AppendEntriesRequest from %s: %s\n", senderAddress, msg.AppendEntriesRequest)
        handleAppendEntriesRequest(*msg)
    case *miniraft.Raft_AppendEntriesResponse:
        fmt.Printf("Received AppendEntriesResponse from %s: %s\n", senderAddress, msg.AppendEntriesResponse)
        handleAppendEntriesResponse(*msg, senderAddress)
    case *miniraft.Raft_RequestVoteRequest:
        fmt.Println("Received RequestVoteRequest: ", msg.RequestVoteRequest)
        handleRequestVoteRequest(*msg)
    case *miniraft.Raft_RequestVoteResponse:
        fmt.Println("Received RequestVoteResponse: ", msg.RequestVoteResponse)
        handleRequestVoteResponse(*msg)
    default:
        log.Printf("[handleMessage] Received an unknown type of message: %T", msg)
    }
}
```

Figure 17: handleMessage handles the different types of messages

Refer to Figure 16, the server on its assigned address and starts a timeout timer. When it receives a message it calls handleMessage (refer to Figure 17).

```go
// This function connects the client process to the server
func startClient(serverHostPort string) {
    conn, err := net.ListenPacket("udp", ":0")
    if err ≠ nil {
        log.Fatal(err)
    }

    connection = conn;

    dst, err := net.ResolveUDPAddr("udp", serverHostPort)
    if err ≠ nil {
        log.Fatal(err)
    }
    destinationAddr = dst
}
```

Figure 18: startClient connects the client to the defined address

Refer to Figure 18, the client has a similar setup and connects to the defined server address via UDP.

2.2 Election Messages

As mentioned in Section 1 of this report, a leader would send out a RequestVoteRequest message to all servers in the network when it has timed out (refer to Figure 2). This message would carry these information:

Term: Term incremented by 1,
LastLogIndex: Last Entry Index,
LastLogTerm: Last Term Index,
CandidateName: Address of this server

A voting server upon receiving this message would send a RequestVoteResponse message with information of whether the vote has been granted by the voting server with respect to the specifications highlighted by Ongaro and Ousterhout which have been explained in Section 1 of this report. The message would carry these informations:

If rejected:

Term: Current term of this server,
VoteGranted: False

If granted:

Term: Current term of this server,
VoteGranted: True

It then sends this message through UDP back to the candidate (refer to Figure 3).

2.3 Log Messages

A leader sends an AppendEntriesRequest in two cases, either when it receives a new command from the client or it is sending out its heartbeat to assert its authority in the network. In the latter case, the AppendEntriesRequest would be an empty request without any new entries, but it still contains information about the current term, leader identity, previous log index, previous log term and the leader's last committed index. A heartbeat message would have the following information:

Term: Current term,
LeaderId: Leader's address,
PrevLogIndex: Last Entry Index,
PrevLogTerm: Last Term Index,
Entries: nil,
LeaderCommit: Commit Index of the leader

In the former case, upon receiving a command from a client, a leader would append the command into its own log and send an AppendEntriesRequest requesting other servers to replicate this entry in their logs as well. This message would have the following information:

Term: Current term,
LeaderId: Leader's address,
PrevLogIndex: Last Entry Index of leader,
PrevLogTerm: Last Term Index of leader,
Entries: New log entry,
LeaderCommit: Commit Index of the leader

A server receiving this AppendEntriesRequest from the leader (note that the leader could be old and an invalid leader) would handle it accordingly. This has been described in detail in Section 1 of this report. The receiving server would handle the the append request based on the specifications laid out by Ongaro and Ousterhout and returns either an AppendEntriesResponse that has been rejected or accepted back to the leader. The response would have the following information:

If accepted:

Term: Current term,
Success: True

If rejected:

Term: Current term,
Success: False

If a leader receives an accepted response, it updates the known index that was last replicated by that server and sends any additional entries still not present on that server's log by sending another AppendEntriesRequest. The leader is able to keep track of which index to send next for each server based on its nextIndex array.

If a leader receives a rejected response, it knows that server has denied appending the new entry to its log. This could be due to this leader being an older leader and the response is coming from a new leader with a later term. The response could also have been rejected due to that server not having the previous log required to ensure that the replication of the log is consistent. The leader then decrements the next index to be sent to that server and tries again by sending another AppendEntriesRequest. The above then repeats when a server receives the AppendEntriesRequest message.

3 Heartbeat Mechanism

In this section, the heartbeat mechanism is explained in full details.

3.1 Single Heartbeat

A heartbeat is sent as an empty AppendEntriesRequest (see lines 3 to 10 in Figure 8). The server then broadcast this message to all other servers in the network (refer to Figure 19). An initial heartbeat is sent immediately following a candidate converting into a leader upon a successful election, this allows it to assert authority in the network.

```go
/* Function to broadcast msg to all servers that are in the server list */
func broadcastMessage(message *miniraft.Raft){
    updateServersKnowledge() // To update list of known servers

    // If server is suspended
    if suspended {
        return
    }

    // Send vote Request to other followers
    msg, err := proto.Marshal(message)
    if err != nil {
        log.Fatal("Error when sending message")
    }

    // Iterate over the server addresses
    for _, addr := range servers {
        if addr != serverHostPort {
            dst, err := net.ResolveUDPAddr("udp", addr)
            if err != nil {
                log.Fatal(err)
            }

            // Serialize the message
            _, err = listener.WriteToUDP(msg, dst)
            log.Printf("SendMiniRaftMessage(): sending %s, %d to %s\n", message, len(msg), dst.String())
            if err != nil {
                log.Panicln("Failed to marshal message.", err)
            }

            fmt.Printf("Message sent to %s\n", addr)
        }
    }
}
```

Figure 19: broadcastMessage sends a message to all servers in the network

3.2 Periodic Heartbeat

A heartbeat routine is started when a leader executes SetHeartBeatRoutine (refer to Figure 7). This creates a counter that times out and fires SendHeartBeat every heartbeatInterval (set to 5 seconds in the implementation). There exist a loop in SetHeartBeatRoutine in which a leader would continue to setup this counter. To prevent multiple counters from instantiating, the implementation incorporated a heartbeatTimerIsActive variable that would be set to true when the counter has started and set to false when the heartbeat has been sent. This resets heartbeat counter (see lines 6 to 9 in Figure 7).

When a candidate becomes a leader, it initially sends out a heartbeat to assert its authority and then sets up this heartbeat routine. The heartbeat interval in this implementation is 5 seconds which is short enough as compared to the minimum time allocated for timeouts which is 10 seconds (the interval for timeouts is 10 to 100 seconds chosen randomly in this implementation). This ensures that a leader would continue to assert its authority if it is alive without having other servers timeout and starting an election.

4 Timeouts

This short section explains how new elections are called if heartbeats are not received on time. resetTimer (refer to Figure 1) is a function that resets or starts a the timeout counters when called.

Refer to Figure 16, in handleCommunication the server would listen onto the assigned UDP address. When a server starts, it would start the timeout counter (see lines 16 to 18). Upon receiving data from this UDP address, it would call resetTimer and this resets the timeout counter. This ensures that as long as the server receives a message from another, it would know that a leader is still in authority.

If a leader suspends and is unable to send out heartbeats, a follower would not receive any data and hence would not reset its timeout counter. Upon the expiration of its counter, it fires handleTimeOut which starts the election process with it as the candidate.

5 Progress and Deadlock Freedom

To discuss progress and deadlock freedom, the list of mutexes used shall be kept in view. The implementation for servers contains four mutexes, namely mutex, timerMutex, heartbeatTimerMutex and updateLogMutex.

```
1   // Getter and Setter for server HostPort Address
2   func setServerHostPort(hostPort string){
3       mutex.Lock()
4       defer mutex.Unlock()
5       serverHostPort = hostPort
6   }
7
8   func getServerHostPort() string{
9       mutex.Lock()
10      defer mutex.Unlock()
11      return serverHostPort
12  }
13
```

Figure 20: Getter and Setter functions to obtain server's address

Mutex (variable name) is used in setServerHostPort, getServerHostPort (refer to Figure 20) and handleMessage (refer to Figure 17). As can be seen, setServerHostPort and getServerHostPort will eventually terminate, as it does not wait on any conditions during its execution after obtaining the

mutex lock, and unlock the mutex lock. Additionally, handleMessage would also obtain the lock and release it at the end of its execution. Since handleMessage relies on possibly five other possible functions, handleCommandName, handleAppendEntriesRequest, handleAppendEntriesResponse, handleRequestVoteRequest and handleRequestVoteResponse, it is necessary for these functions to also terminate eventually.

handleCommandName (refer to Figure 9, handleAppendEntriesResponse, handleRequestVoteRequest and handleRequestVoteResponse does not rely on any conditions that would cause them to hold-and-wait. All possible execution paths would eventually lead to their complete execution.

handleAppendEntriesRequest similarly does not halt execution at any point due to an unsatisfiable condition, it proceeds and would eventually terminate. This function also relies on UpdateLogFile which is called when updating/writing to the log file. UpdateLogFile relies on updateLogMutex mutex which could lead to a possible halt condition (refer to Figure 21). However, since UpdateLogFile is called in a separate go routine, handleAppendEntriesRequst will still eventually terminate.

```go
/* This function updates the server's log (this should be ran concurrently) */
func UpdateLogFile(){
    updateLogMutex.Lock()
    defer updateLogMutex.Unlock()

    serverHostPortSafe := strings.Replace(serverHostPort, ":", "-", -1)

    // Construct the file name based on the server's host and port.
    fileName := fmt.Sprintf("../logs/%s.log", serverHostPortSafe)

    // Open the file in write mode. Create it if it does not exist.
    file, err := os.OpenFile(fileName, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0644)
    if err != nil {
        log.Fatalf("Failed to open log file: %v", err)
    }
    defer file.Close()


    // Construct the log entry.
    for i := 0; i < commitIndex; i++ {
        if i > len(logs) - 1 {
            break
        }
        var logEntry = logs[i];
        logEntryStr := fmt.Sprintf("%d,%d,%s\n", logEntry.GetTerm(),
        logEntry.GetIndex(), logEntry.GetCommandName())

        // Write the log entry to the file.
        if _, err := file.WriteString(logEntryStr); err != nil {
            log.Fatalf("Failed to write to log file: %v", err)
        }
    }
}
```

Figure 21: updateLogFile is used to write and update the log file for the server

Since all the functions that handleMessage rely on would eventually terminate, progress and

deadlock freedom have been achieved for these three processes that rely on mutex (variable name).

timerMutex is used in setupLeader and resetTimer. In setupLeader, the timerMutex is obtained and released after complete execution of lines 19 to 29 in Figure 5. Lines 19 to 29 do not depend on any conditions that can lead to its waiting for resources and hence would complete its execution without waiting. Hence, setupLeader would eventually complete its execution and unlock timerMutex.

resetTimer obtains the timerMutex lock and starts the time out counter and finishes its execution. Again, at no point does resetTimer halt its execution and wait for any resource besides the timerMutex lock. As such, resetTimer would also eventually complete its execution and unlock timerMutex lock.

At lines 21 to 25 in Figure 1, the time out counter executes these statements when the timer has expired. It calls handleTimeOut which depends on resetTimer which as shown above would finish its execution eventually without waiting. Since handleTimeOut would also finish its execution without waiting for any resources, it is safe to state that lines 21 to 25 would finish its execution without waiting for any resources other than the timerMutex lock.

As such, progress and deadlock freedom has also been achieved for the two processes that rely on timerMutex.

heartbeatTimerMutex is used in SetHeartBeatRoutine and SendHeartBeat. SetHeartBeatRoutine obtains the heartbeatTimerMutex before executing lines 6 to 9 in Figure 7. Since lines 6 to 9 does not halt execution, there exist a time interval when this go routine would not be holding onto the heartbeatTimerMutex lock.

SendHeartBeat also relies on heartbeatTimerMutex lock. Since its execution after obtaining the heartbeatTimerMutex lock can be executed without waiting (see line 21 in Figure 8), this function would also eventually release the heartbeatTimerMutex lock.

As such, progress and deadlock freedom has also been achieved for the two processes that rely on heartbeatTimerMutex.

Finally, updateLogMutex is only used in UpdateLogFile (refer to Figure 21). Since this function does not rely on any other resources that can cause it to hold-and-wait, the function would unlock the updateLogMutex lock eventually.

From the above explanations, it is safe to conclude that the implementation ensures progress and deadlock freedom.

A side note to look into the client of this implementation. The client only relies on mutex, mutex lock. This lock is used in setServerHostPort, getServerHostPort, exitClient and sendCommand. These functions does not wait on any conditions, and hence would complete its execution and unlock the mutex lock. As such, the client of this implementation also achieves progress and deadlock freedom.

List of Figures

1 resetTimer starts/restarts the timer for servers in the follower and candidate state . . . 1
2 handleTimeOut checks for timeout of a client and if so turns the client into a candidate to run for leader election . . . 2
3 handleRequestVoteRequest() handles the procedures to respond to a vote request from a candidate server . . . 3
4 handleRequestVoteResponse() handles the procedures to detect the result of the election 4
5 setupLeader() initialises and setups the conversion from candidate to leader state . . . 4
6 If the candidate receives an AppendEntriesRequest/heartbeat message, it would convert into a Follower . . . 5
7 SetHeartBeatRoutine sets up a timer to send out heartbeat periodically . . . 6
8 SendHeartBeat sends out an empty AppendEntriesRequest message to all servers in the network . . . 6
9 handleCommandName handles an incoming message from a client . . . 8
10 Snippet of HandleAppendEntriesRequest, illustrated to explain a client's response to receiving an AppendEntriesRequest message . . . 9
11 Snippet of HandleAppendEntriesRequest, illustrated to explain a client's response to receiving an AppendEntriesRequest message . . . 10
12 Snippet of HandleAppendEntriesRequest, illustrated to explain a client's response to receiving an AppendEntriesRequest message . . . 11
13 Snippet of HandleAppendEntriesResponse, illustrated to explain a leader's response to receiving an AppendEntriesResponse message . . . 12
14 Snippet of HandleAppendEntriesResponse, illustrated to explain a leader's response to receiving an AppendEntriesResponse message . . . 14
15 startServer is executed once when the server starts up . . . 15
16 handleCommunication starts the server listening to its assigned address . . . 16
17 handleMessage handles the different types of messages . . . 17
18 startClient connects the client to the defined address . . . 18
19 broadcastMessage sends a message to all servers in the network . . . 21
20 Getter and Setter functions to obtain server's address . . . 22
21 updateLogFile is used to write and update the log file for the server . . . 23
22 Figure 2 of Ongaro and Ousterhout's report [1] . . . IV

Nomenclature

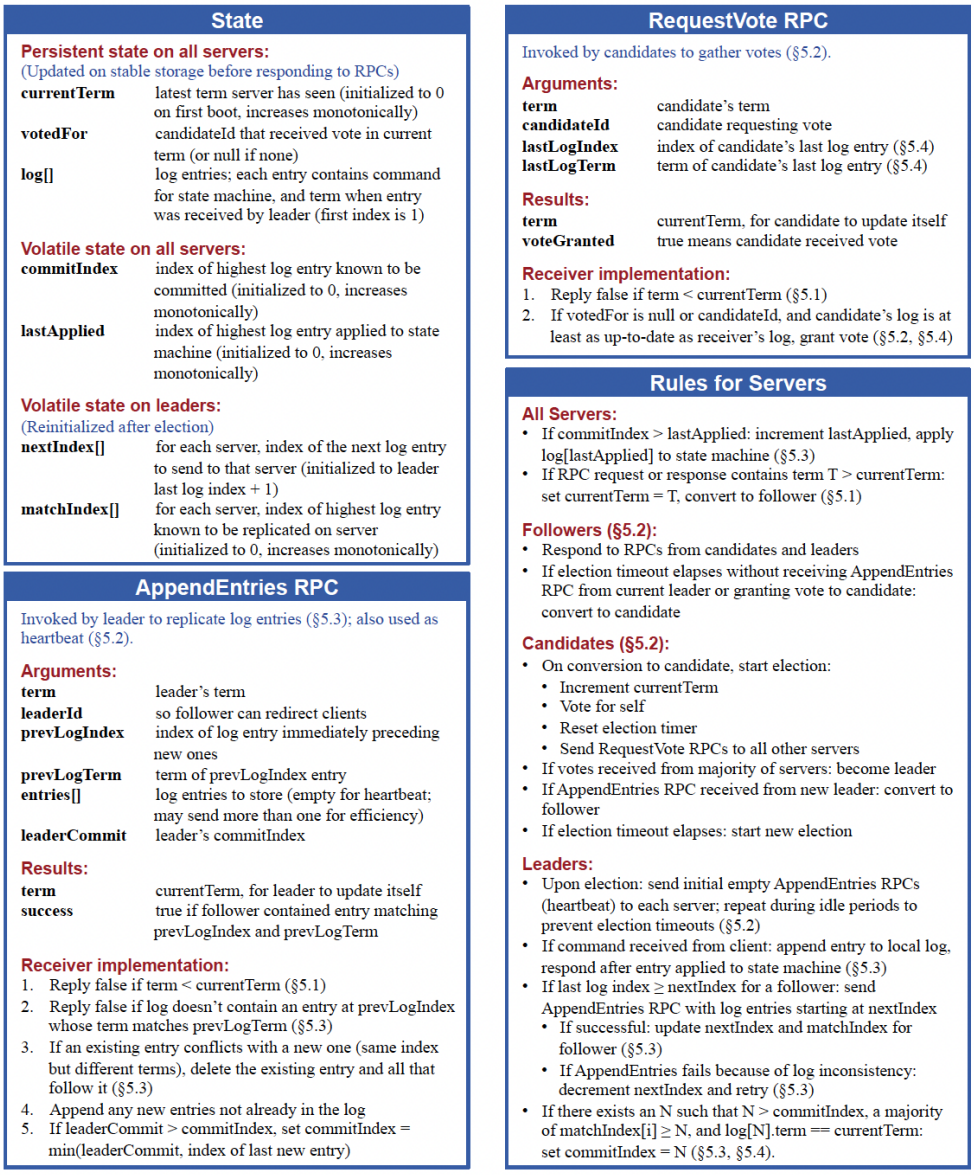
RPC Remote Procedural Call

UDP User Datagram Protocol

References

[1] D. Ongaro and J. Ousterhout, “In search of an understandable consensus algorithm,” in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, (Philadelphia, PA), pp. 305–319, USENIX Association, June 2014.

6 Appendix A



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.

Figure 22: Figure 2 of Ongaro and Ousterhout's report [1]