RAFT Code Explanation

Leader Election Loops

This is the crucial and first most part of RAFT algorithm where we select a leader for our cluster. This is implemented using the following loops

```go
// execute different processes based on the role of this Raft node.
func (rf *Raft) mainLoop() {

    for !rf.killed() {
        /*++++++++++++++++++++++++++++++++++++++++++++*/
        rf.mu.Lock()
        r := rf.role
        rf.mu.Unlock()
        /*------------------------------------------*/
        switch r {
        case Follower:
            rf.followerLoop()
        case Candidate:
            rf.candidateLoop()
        case Leader:
            rf.leaderLoop()
        }
    }
}
```

```go
func (rf *Raft) followerLoop() {

    for !rf.killed() {

        rf.mu.Lock()

        // #region set a new Trigger and kick off…

        rf.trigger.Wait()


        rf.mu.Lock()
        switch {
        case rf.trigger.Elapsed: // timer naturally elapses, turns to Candidate…
        default: // stays as Follower, set a new Trigger in the next round…
        }
        rf.mu.Unlock()

    }
}
```

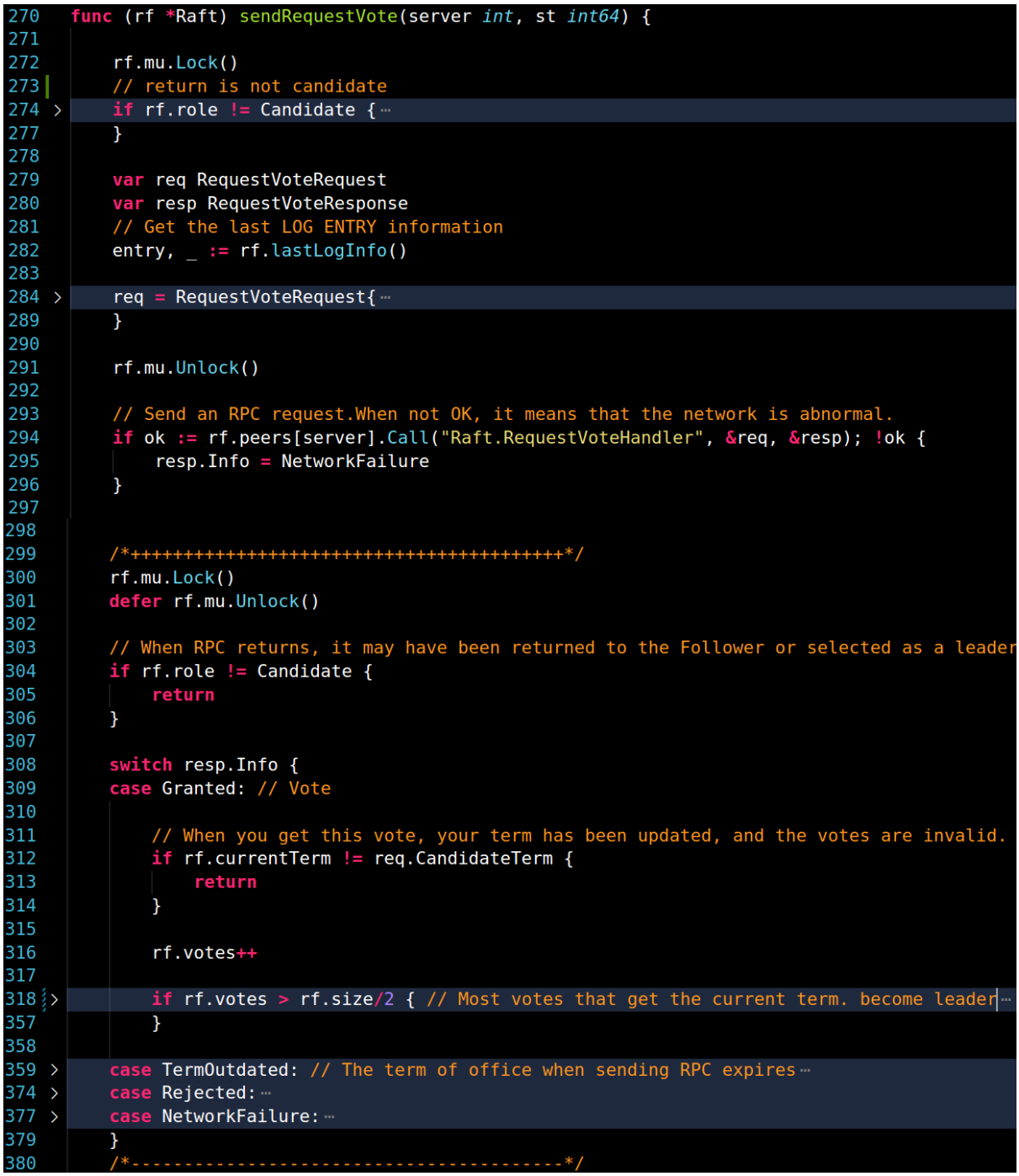
```go
func (rf *Raft) leaderLoop() {

    // periodically send heartbeats
    /*------------------------------------------*/
    for !rf.killed() {

        /*++++++++++++++++++++++++++++++++++++++++++++*/
        rf.mu.Lock()
        if rf.role != Leader {
            rf.mu.Unlock()
            return
        }
        rf.mu.Unlock()
        /*------------------------------------------*/

        // concurrently send heartbeats
        for i := 0; i < rf.size; i++ {
            if i == rf.me {
                continue
            }

            go rf.sendHeartBeat(i)
        }

        // after sending heartbeats, sleep Leader for a while
        sleeper := time.NewTimer(HeartBeatInterval * time.Millisecond)
        <-sleeper.C
    }
}
```

```go
func (rf *Raft) candidateLoop() {

    for !rf.killed() {
        Debug(rf, "start new election")

        // region reset the timer for election end…

        for i := 0; i < rf.size; i++ {
            if i == rf.me {
                continue
            }

            go rf.sendRequestVote(i, rf.trigger.StartTime)
        }


        // Concurrency RequestVote RPCs

        rf.trigger.Wait()

        /*++++++++++++++++++++++++++++++++++++++++++++*/
        // region :Judging whether the role changes in the next cycle…
    }
}
```

Vote Request RPC

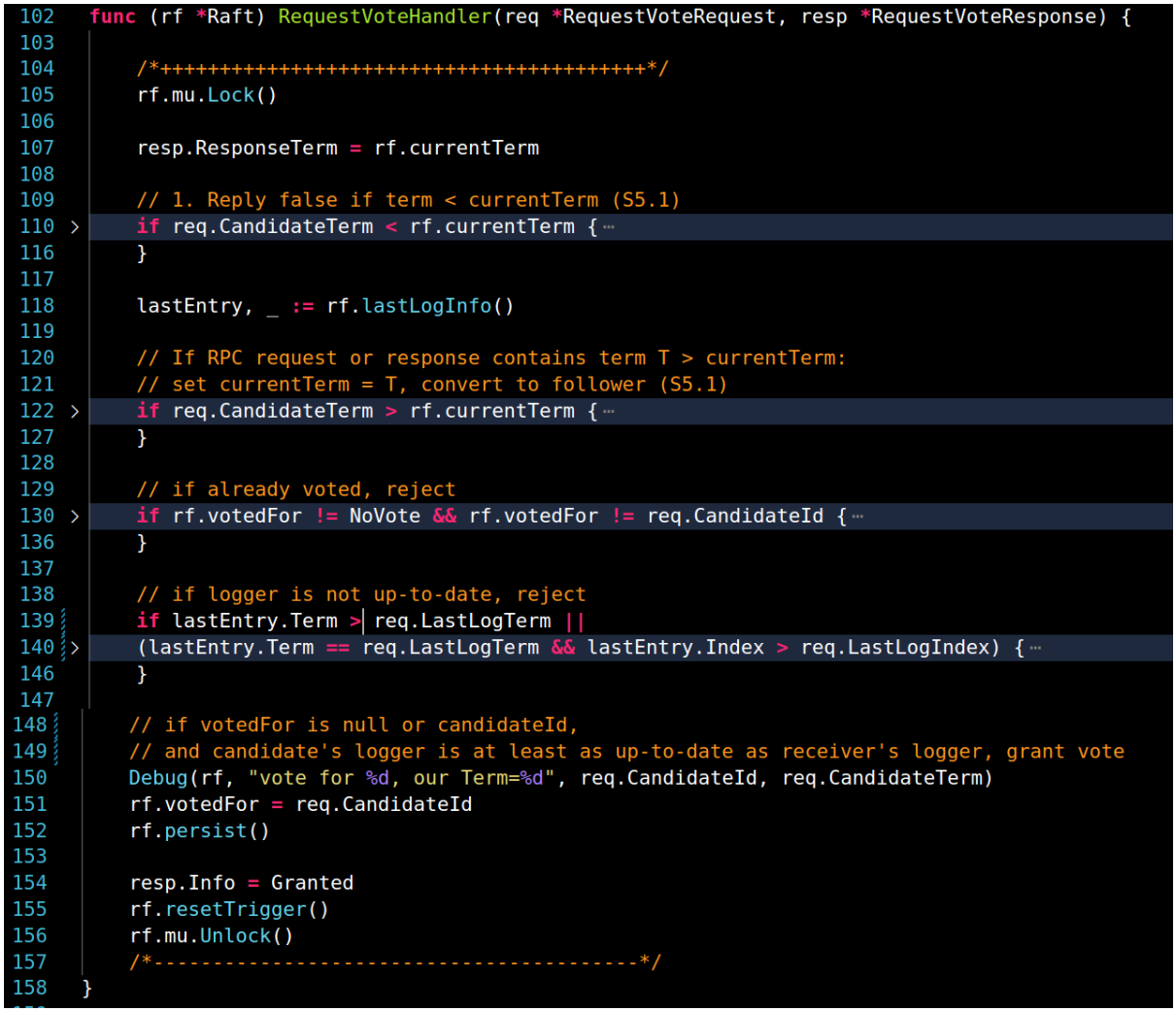
RPC Handler

Sends vote requests async, and wait for response, becomes leader is gained majority of votes. If got minority of votes and no leader then re-election is done and if new leader is there, then reverts to follower



```
func (rf *Raft) sendRequestVote(server int, st int64) {

    rf.mu.Lock()
    // return is not candidate
    if rf.role != Candidate {…
    }

    var req RequestVoteRequest
    var resp RequestVoteResponse
    // Get the last LOG ENTRY information
    entry, _ := rf.lastLogInfo()

    req = RequestVoteRequest{…
    }

    rf.mu.Unlock()

    // Send an RPC request.When not OK, it means that the network is abnormal.
    if ok := rf.peers[server].Call("Raft.RequestVoteHandler", &req, &resp); !ok {
        resp.Info = NetworkFailure
    }


    /*++++++++++++++++++++++++++++++++++++++++++*/
    rf.mu.Lock()
    defer rf.mu.Unlock()

    // When RPC returns, it may have been returned to the Follower or selected as a leader
    if rf.role != Candidate {
        return
    }

    switch resp.Info {
    case Granted: // Vote

        // When you get this vote, your term has been updated, and the votes are invalid.
        if rf.currentTerm != req.CandidateTerm {
            return
        }

        rf.votes++

        if rf.votes > rf.size/2 { // Most votes that get the current term. become leader…
        }

    case TermOutdated: // The term of office when sending RPC expires…
    case Rejected:…
    case NetworkFailure:…
    }
    /*------------------------------------------*/
```

RPC function

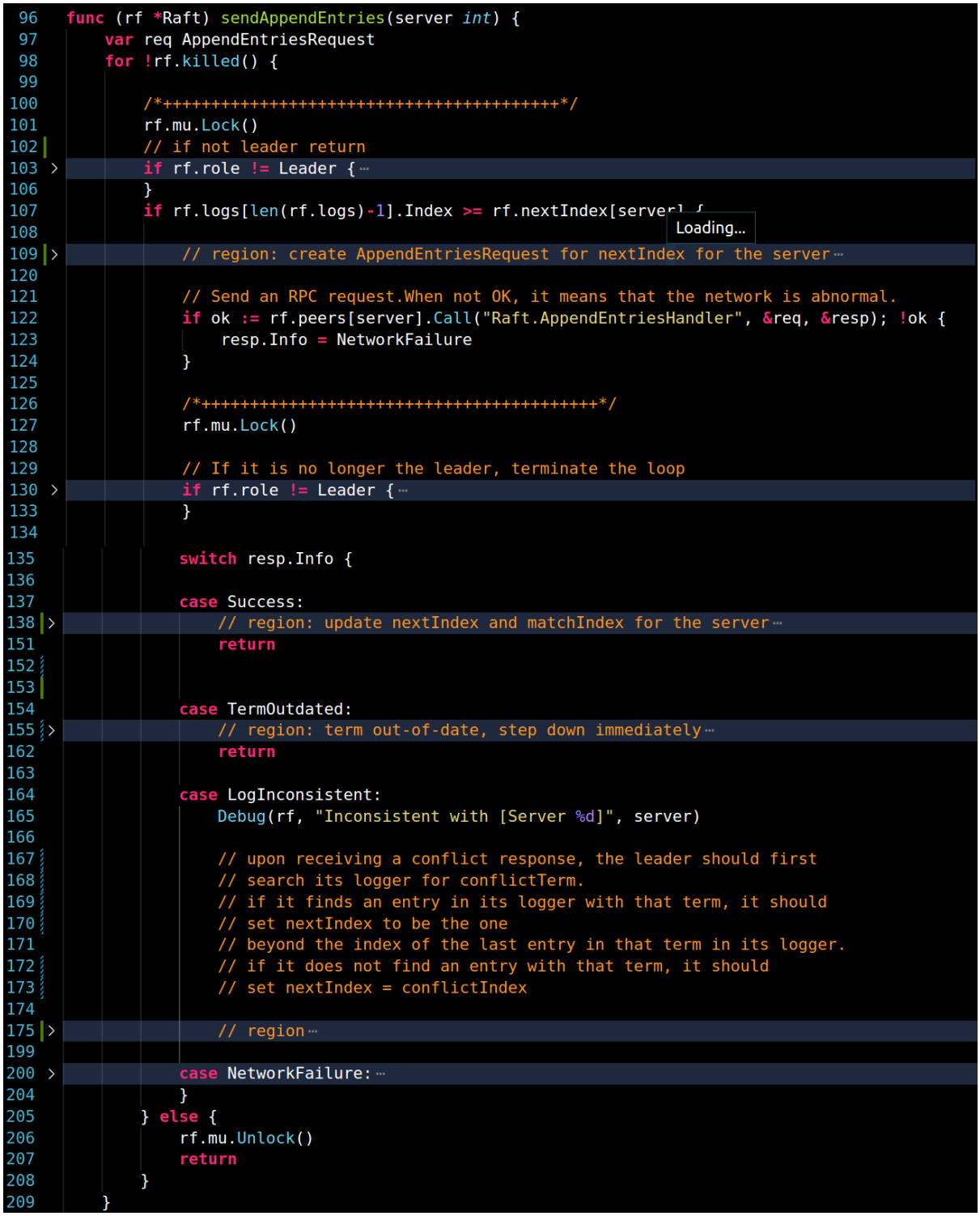


```
102  func (rf *Raft) RequestVoteHandler(req *RequestVoteRequest, resp *RequestVoteResponse) {
103
104      /*++++++++++++++++++++++++++++++++++++++++*/
105      rf.mu.Lock()
106
107      resp.ResponseTerm = rf.currentTerm
108
109      // 1. Reply false if term < currentTerm (S5.1)
110  >   if req.CandidateTerm < rf.currentTerm {…
116      }
117
118      lastEntry, _ := rf.lastLogInfo()
119
120      // If RPC request or response contains term T > currentTerm:
121      // set currentTerm = T, convert to follower (S5.1)
122  >   if req.CandidateTerm > rf.currentTerm {…
127      }
128
129      // if already voted, reject
130  >   if rf.votedFor != NoVote && rf.votedFor != req.CandidateId {…
136      }
137
138      // if logger is not up-to-date, reject
139      if lastEntry.Term > req.LastLogTerm ||
140  >   (lastEntry.Term == req.LastLogTerm && lastEntry.Index > req.LastLogIndex) {…
146      }
147
148      // if votedFor is null or candidateId,
149      // and candidate's logger is at least as up-to-date as receiver's logger, grant vote
150      Debug(rf, "vote for %d, our Term=%d", req.CandidateId, req.CandidateTerm)
151      rf.votedFor = req.CandidateId
152      rf.persist()
153
154      resp.Info = Granted
155      rf.resetTrigger()
156      rf.mu.Unlock()
157      /*----------------------------------------*/
158  }
```

Append Entries RPC

RPC Handler

Sends RPC request and updates the last match index variables in leader if logs do not match for re-sending correct values in next cycle.



RPC Function

Follower accepts new logs from leader and appends if previous ones are matching else report it to the leader.

```go
func (rf *Raft) AppendEntriesHandler(req *AppendEntriesRequest, resp *AppendEntriesResponse) {

    /*+++++++++++++++++++++++++++++++++++++++*/
    rf.mu.Lock()
    defer rf.printLog()

    resp.ResponseTerm = rf.currentTerm

    // 1. Reply false if term < currentTerm (S5.1)
    if req.LeaderTerm < rf.currentTerm {…
    }

    // reset the Trigger
    rf.resetTrigger()

    // If RPC request or response contains term T > currentTerm:
    // set currentTerm = T, convert to follower (S5.1)
    if req.LeaderTerm > rf.currentTerm {…
    }

    // The original text was not discussed
// Positive value indicates that the index entry
// greater than equal to Len represents beyond
    sliceIdx := req.PrevLogIndex - rf.offset

    switch {

    // If a follower does not have prevLogIndex in its log,
    // it should return with conflictIndex = len(log) and conflictTerm = None.
    case sliceIdx >= len(rf.logs):…
    case sliceIdx == -1:
    // 2. Reply false if logger doesn't contain an entry at prevLogIndex
    // whose term matches prevLogTerm (S5.3)
    default:…
    }

    resp.Info = Success

    // 3. If an existing entry conflicts with a new one (same index
    // but different terms), delete the existing entry and all that
    // follow it (S5.3)
    // 4. Append any new entries not already in the log
    i := sliceIdx + 1
    j := 0

    for j < len(req.Entries) {
        if i == len(rf.logs) {
            rf.logs = append(rf.logs, req.Entries[j])
        } else if rf.logs[i].Term != req.Entries[j].Term {
            rf.logs = rf.logs[:i]
            rf.logs = append(rf.logs, req.Entries[j])
        }
        i++
        j++
    }
    rf.persist()

    // 5. If leaderCommit > commitIndex, set commitIndex =
    // min(leaderCommit, index of last new entry)
    rf.receiverTryUpdateCommitIndex(req)
    rf.mu.Unlock()
    /*---------------------------------------*/
}
```