

RAFT Code Explanation

[Leader Election Loops](#)

[Vote Request RPC](#)

[RPC Handler](#)

[RPC function](#)

[Append Entries RPC](#)

[RPC Handler](#)

[RPC Function](#)

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

```
61 // execute different processes based on the role of this Raft node.
62 func (rf *Raft) mainLoop() {
63
64     for !rf.killed() {
65         /*+++++++*/
66         rf.mu.Lock()
67         r := rf.role
68         rf.mu.Unlock()
69         /*-----*/
70         switch r {
71             case Follower:
72                 rf.followerLoop()
73             case Candidate:
74                 rf.candidateLoop()
75             case Leader:
76                 rf.leaderLoop()
77         }
78     }
79 }
```

```
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()
    }
}
```

```
174 func (rf *Raft) leaderLoop() {
175
176     // periodically send heartbeats
177     /*-----*/
178     for !rf.killed() {
179         /*+++++++*/
180         rf.mu.Lock()
181         if rf.role != Leader {
182             rf.mu.Unlock()
183             return
184         }
185         rf.mu.Unlock()
186         /*-----*/
187         // concurrently send heartbeats
188         for i := 0; i < rf.size; i++ {
189             if i == rf.me {
190                 continue
191             }
192             go rf.sendHeartBeat(i)
193         }
194         // after sending heartbeats, sleep Leader for a while
195         sleeper := time.NewTimer(HeartBeatInterval * time.Millisecond)
196         <-sleeper.C
197     }
198 }
```

```
120 func (rf *Raft) candidateLoop() {
121
122     for !rf.killed() {
123         Debug(rf, "start new election")
124         // region reset the timer for election end--
125         rf.trigger.Reset()
126         for i := 0; i < rf.size; i++ {
127             if i == rf.me {
128                 continue
129             }
130             go rf.sendRequestVote(i, rf.trigger.StartTime)
131         }
132         // Concurrency RequestVote RPCs
133         rf.trigger.Wait()
134         /*+++++++*/
135         // region :Judging whether the role changes in the next cycle--
136         rf.mu.Lock()
137         if rf.role == Candidate {
138             rf.mu.Unlock()
139             continue
140         }
141         rf.mu.Unlock()
142     }
143 }
```

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

```
270 func (rf *Raft) sendRequestVote(server int, st int64) {
271     rf.mu.Lock()
272     // return is not candidate
273     if rf.role != Candidate { ...
274 }
275     }
276
277     var req RequestVoteRequest
278     var resp RequestVoteResponse
279     // Get the last LOG ENTRY information
280     entry, _ := rf.lastLogInfo()
281
282     req = RequestVoteRequest{ ...
283 }
284     }
285
286     rf.mu.Unlock()
287
288     // Send an RPC request. When not OK, it means that the network is abnormal.
289     if ok := rf.peers[server].Call("Raft.RequestVoteHandler", &req, &resp); !ok {
290         resp.Info = NetworkFailure
291     }
292
293     /*+++++*/
294     rf.mu.Lock()
295     defer rf.mu.Unlock()
296
297     // When RPC returns, it may have been returned to the Follower or selected as a leader
298     if rf.role != Candidate {
299         return
300     }
301
302     switch resp.Info {
303     case Granted: // Vote
304
305         // When you get this vote, your term has been updated, and the votes are invalid.
306         if rf.currentTerm != req.CandidateTerm {
307             return
308         }
309
310         rf.votes++
311
312         if rf.votes > rf.size/2 { // Most votes that get the current term. become leader
313 }
314     }
315
316     case TermOutdated: // The term of office when sending RPC expires
317 }
318     case Rejected: ...
319     case NetworkFailure: ...
320     }
321     /*-----*/
```

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 { ...
111     }
112
113     lastEntry, _ := rf.lastLogInfo()
114
115     // If RPC request or response contains term T > currentTerm:
116     // set currentTerm = T, convert to follower (S5.1)
117 > if req.CandidateTerm > rf.currentTerm { ...
118     }
119
120     // if already voted, reject
121 > if rf.votedFor != NoVote && rf.votedFor != req.CandidateId { ...
122     }
123
124     // if logger is not up-to-date, reject
125     if lastEntry.Term > req.LastLogTerm ||
126 > (lastEntry.Term == req.LastLogTerm && lastEntry.Index > req.LastLogIndex) { ...
127     }
128
129     // if votedFor is null or candidateId,
130     // and candidate's logger is at least as up-to-date as receiver's logger, grant vote
131     Debug(rf, "vote for %d, our Term=%d", req.CandidateId, req.CandidateTerm)
132     rf.votedFor = req.CandidateId
133     rf.persist()
134
135     resp.Info = Granted
136     rf.resetTrigger()
137     rf.mu.Unlock()
138     /*-----*/
139 }

```

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.

```

96 func (rf *Raft) sendAppendEntries(server int) {
97     var req AppendEntriesRequest
98     for !rf.killed() {
99
100         /*+++++*/
101         rf.mu.Lock()
102         // if not leader return
103         if rf.role != Leader { ...
104     }
105     if rf.logs[len(rf.logs)-1].Index >= rf.nextIndex[server] {
106         // region: create AppendEntriesRequest for nextIndex for the server...
107
108         // Send an RPC request. When not OK, it means that the network is abnormal.
109         if ok := rf.peers[server].Call("Raft.AppendEntriesHandler", &req, &resp); !ok {
110             resp.Info = NetworkFailure
111         }
112
113         /*+++++*/
114         rf.mu.Lock()
115
116         // If it is no longer the leader, terminate the loop
117         if rf.role != Leader { ...
118     }
119
120     switch resp.Info {
121     case Success:
122         // region: update nextIndex and matchIndex for the server...
123         return
124
125     case TermOutdated:
126         // region: term out-of-date, step down immediately...
127         return
128
129     case LogInconsistent:
130         Debug(rf, "Inconsistent with [Server %d]", server)
131
132         // upon receiving a conflict response, the leader should first
133         // search its logger for conflictTerm.
134         // if it finds an entry in its logger with that term, it should
135         // set nextIndex to be the one
136         // beyond the index of the last entry in that term in its logger.
137         // if it does not find an entry with that term, it should
138         // set nextIndex = conflictIndex
139
140         // region...
141
142     case NetworkFailure: ...
143     }
144     } else {
145         rf.mu.Unlock()
146         return
147     }
148 }
149 }

```

RPC Function

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

```

5 func (rf *Raft) AppendEntriesHandler(req *AppendEntriesRequest, resp *AppendEntriesResponse) {
6
7     /*+++++*/
8     rf.mu.Lock()
9     defer rf.printLog()
10
11     resp.ResponseTerm = rf.currentTerm
12
13     // 1. Reply false if term < currentTerm (S5.1)
14 > if req.LeaderTerm < rf.currentTerm { ...
18     }
19
20     // reset the Trigger
21     rf.resetTrigger()
22
23     // If RPC request or response contains term T > currentTerm:
24     // set currentTerm = T, convert to follower (S5.1)
25 > if req.LeaderTerm > rf.currentTerm { ...
29     }
30
31     // The original text was not discussed
32 // Positive value indicates that the index entry
33 // greater than equal to Len represents beyond
34 sliceIdx := req.PrevLogIndex - rf.offset
35
36     switch {
37
38     // If a follower does not have prevLogIndex in its log,
39     // it should return with conflictIndex = len(log) and conflictTerm = None.
40 > case sliceIdx >= len(rf.logs): ...
47     case sliceIdx == -1:
50 // 2. Reply false if logger doesn't contain an entry at prevLogIndex
51 // whose term matches prevLogTerm (S5.3)
52 > default: ...
66     }
67
68     resp.Info = Success
69
70     // 3. If an existing entry conflicts with a new one (same index
71     // but different terms), delete the existing entry and all that
72     // follow it (S5.3)
73     // 4. Append any new entries not already in the log
74     i := sliceIdx + 1
75     j := 0
76
77     for j < len(req.Entries) {
78         if i == len(rf.logs) {
79             rf.logs = append(rf.logs, req.Entries[j])
80         } else if rf.logs[i].Term != req.Entries[j].Term {
81             rf.logs = rf.logs[:i]
82             rf.logs = append(rf.logs, req.Entries[j])
83         }
84         i++
85         j++
86     }
87     rf.persist()
88
89     // 5. If leaderCommit > commitIndex, set commitIndex =
90     // min(leaderCommit, index of last new entry)
91     rf.receiverTryUpdateCommitIndex(req)
92     rf.mu.Unlock()
93     /*-----*/
94 }

```