

thuduyen07 /
raft

<> Code

Pull requests

Actions

Projects

Wiki

Security

Insights



raft / part1 / raft.go



thuduyen07 add doc for raft ✓

2 minutes ago



405 lines (354 loc) · 10.2 KB

```
1 // Core Raft implementation - Consensus Module.
2 //
3 // Eli Bendersky [https://eli.thegreenplace.net]
4 // This code is in the public domain.
5 package raft
6
7 import (
8     "fmt"
9     "log"
10    "math/rand"
11    "os"
12    "sync"
13    "time"
14 )
15
16 const DebugCM = 1
17
18 /*
19     cấu trúc log gồm:
20     - một mục Command (lệnh được gửi)
21     - Term (nhiệm kỳ hiện hành)
22 */
23 type LogEntry struct {
24     Command interface{}
25     Term    int
26 }
27
28 /*
29     Trạng thái của một Consensus Module
30 */
31 type CMState int
32
33 const (
```

```
34         Follower CMState = iota
35         Candidate
36         Leader
37         Dead
38     )
39
40     func (s CMState) String() string {
41         switch s {
42         case Follower:
43             return "Follower"
44         case Candidate:
45             return "Candidate"
46         case Leader:
47             return "Leader"
48         case Dead:
49             return "Dead"
50         default:
51             panic("unreachable")
52         }
53     }
54
55     // ConsensusModule (CM) implements a single node of Raft consensus.
56     type ConsensusModule struct {
57         // mu protects concurrent access to a CM.
58         mu sync.Mutex
59
60         // id is the server ID of this CM.
61         id int
62
63         // peerIds lists the IDs of our peers in the cluster.
64         peerIds []int
65
66         // server is the server containing this CM. It's used to issue RPC calls
67         // to peers.
68         server *Server
69
70         // Persistent Raft state on all servers
71         currentTerm int
72         votedFor    int
73         log          []LogEntry
74
75         // Volatile Raft state on all servers
76         state          CMState
77         electionResetEvent time.Time
78     }
79
80     // NewConsensusModule creates a new CM with the given ID, list of peer IDs and
81     // server. The ready channel signals the CM that all peers are connected and
82     // it's safe to start its state machine.
```

```

83  ✓ func NewConsensusModule(id int, peerIds []int, server *Server, ready <-chan interface{}) *Con
84      cm := new(ConsensusModule)
85      cm.id = id
86      cm.peerIds = peerIds
87      cm.server = server
88      cm.state = Follower
89      cm.votedFor = -1
90
91      go func() {
92          // The CM is quiescent until ready is signaled; then, it starts a countdown
93          // for leader election.
94          <-ready
95          cm.mu.Lock()
96          cm.electionResetEvent = time.Now()
97          cm.mu.Unlock()
98          cm.runElectionTimer()
99      }()
100
101      return cm
102  }
103
104  // Report reports the state of this CM.
105  ✓ func (cm *ConsensusModule) Report() (id int, term int, isLeader bool) {
106      cm.mu.Lock()
107      defer cm.mu.Unlock()
108      return cm.id, cm.currentTerm, cm.state == Leader
109  }
110
111  // Stop stops this CM, cleaning up its state. This method returns quickly, but
112  // it may take a bit of time (up to ~election timeout) for all goroutines to
113  // exit.
114  ✓ func (cm *ConsensusModule) Stop() {
115      cm.mu.Lock()
116      defer cm.mu.Unlock()
117      cm.state = Dead
118      cm.dlog("becomes Dead")
119  }
120
121  // dlog logs a debugging message if DebugCM > 0.
122  ✓ func (cm *ConsensusModule) dlog(format string, args ...interface{}) {
123      if DebugCM > 0 {
124          format = fmt.Sprintf("[%d] ", cm.id) + format
125          log.Printf(format, args...)
126      }
127  }
128
129  // See figure 2 in the paper.
130  ✓ type RequestVoteArgs struct {
131      Term      int

```

```
132         CandidateId int
133         LastLogIndex int
134         LastLogTerm int
135     }
136
137     type RequestVoteReply struct {
138         Term int
139         VoteGranted bool
140     }
141
142     // RequestVote RPC.
143     ✓ func (cm *ConsensusModule) RequestVote(args RequestVoteArgs, reply *RequestVoteReply) error {
144         cm.mu.Lock()
145         defer cm.mu.Unlock()
146         if cm.state == Dead {
147             return nil
148         }
149         cm.dlog("RequestVote: %+v [currentTerm=%d, votedFor=%d]", args, cm.currentTerm, cm.votedFor)
150
151         if args.Term > cm.currentTerm {
152             cm.dlog("... term out of date in RequestVote")
153             cm.becomeFollower(args.Term)
154         }
155
156         if cm.currentTerm == args.Term &&
157             (cm.votedFor == -1 || cm.votedFor == args.CandidateId) {
158             reply.VoteGranted = true
159             cm.votedFor = args.CandidateId
160             cm.electionResetEvent = time.Now()
161         } else {
162             reply.VoteGranted = false
163         }
164         reply.Term = cm.currentTerm
165         cm.dlog("... RequestVote reply: %+v", reply)
166         return nil
167     }
168
169     // See figure 2 in the paper.
170     ✓ type AppendEntriesArgs struct {
171         Term int
172         LeaderId int
173
174         PrevLogIndex int
175         PrevLogTerm int
176         Entries []LogEntry
177         LeaderCommit int
178     }
179
180     type AppendEntriesReply struct {
```

```

181         Term    int
182         Success bool
183     }
184
185     func (cm *ConsensusModule) AppendEntries(args AppendEntriesArgs, reply *AppendEntriesReply) error {
186         cm.mu.Lock()
187         defer cm.mu.Unlock()
188         if cm.state == Dead {
189             return nil
190         }
191         cm.dlog("AppendEntries: %+v", args)
192
193         if args.Term > cm.currentTerm {
194             cm.dlog("... term out of date in AppendEntries")
195             cm.becomeFollower(args.Term)
196         }
197
198         reply.Success = false
199         if args.Term == cm.currentTerm {
200             if cm.state != Follower {
201                 cm.becomeFollower(args.Term)
202             }
203             cm.electionResetEvent = time.Now()
204             reply.Success = true
205         }
206
207         reply.Term = cm.currentTerm
208         cm.dlog("AppendEntries reply: %+v", *reply)
209         return nil
210     }
211
212     /*
213     Phương thức của đối tượng ConsensusModule dùng để triển khai bộ đếm thời gian
214     cho việc bắt đầu một cuộc bầu cử mới. Ở đây đang đặt là 300ms
215     */
216     // electionTimeout generates a pseudo-random election timeout duration.
217     func (cm *ConsensusModule) electionTimeout() time.Duration {
218         // If RAFT_FORCE_MORE_REELECTION is set, stress-test by deliberately
219         // generating a hard-coded number very often. This will create collisions
220         // between different servers and force more re-elections.
221         if len(os.Getenv("RAFT_FORCE_MORE_REELECTION")) > 0 && rand.Intn(3) == 0 {
222             return time.Duration(300) * time.Millisecond
223         } else {
224             return time.Duration(300+rand.Intn(300)) * time.Millisecond
225         }
226     }
227
228     /*
229     Hàm này tạo ra một election timer để đếm ngược đến thời gian bầu cử

```

```

230         và chuyển trạng thái của CM thành Candidate nếu cần.
231     */
232     // runElectionTimer implements an election timer. It should be launched whenever
233     // we want to start a timer towards becoming a candidate in a new election.
234     //
235     // This function is blocking and should be launched in a separate goroutine;
236     // it's designed to work for a single (one-shot) election timer, as it exits
237     // whenever the CM state changes from follower/candidate or the term changes.
238     func (cm *ConsensusModule) runElectionTimer() {
239         timeoutDuration := cm.electionTimeout()
240         cm.mu.Lock()
241         termStarted := cm.currentTerm

```

[raft](#) / [part1](#) / [raft.go](#)
[↑ Top](#)

Code

Blame

Raw



```

238     func (cm *ConsensusModule) runElectionTimer() {
239
240         */
241         // This loops until either:
242         // - we discover the election timer is no longer needed, or
243         // - the election timer expires and this CM becomes a candidate
244         // In a follower, this typically keeps running in the background for the
245         // duration of the CM's lifetime.
246         ticker := time.NewTicker(100 * time.Millisecond)
247         defer ticker.Stop()
248         for {
249             <-ticker.C
250
251             cm.mu.Lock()
252             if cm.state != Candidate && cm.state != Follower {
253                 cm.dlog("in election timer state=%s, bailing out", cm.state)
254                 cm.mu.Unlock()
255                 return
256             }
257
258             if termStarted != cm.currentTerm {
259                 cm.dlog("in election timer term changed from %d to %d, bailing out",
260                     cm.mu.Unlock()
261                     return
262             }
263
264             // Start an election if we haven't heard from a leader or haven't voted for
265             // someone for the duration of the timeout.
266             if elapsed := time.Since(cm.electionResetEvent); elapsed >= timeoutDuration {
267                 cm.startElection()
268                 cm.mu.Unlock()
269                 return
270             }
271
272             // Start an election if we haven't heard from a leader or haven't voted for
273             // someone for the duration of the timeout.
274             if elapsed := time.Since(cm.electionResetEvent); elapsed >= timeoutDuration {
275                 cm.startElection()
276                 cm.mu.Unlock()
277                 return
278             }

```

```

279             cm.mu.Unlock()
280         }
281     }
282
283     // startElection starts a new election with this CM as a candidate.
284     // Expects cm.mu to be locked.
285     func (cm *ConsensusModule) startElection() {
286         cm.state = Candidate
287         cm.currentTerm += 1
288         savedCurrentTerm := cm.currentTerm
289         cm.electionResetEvent = time.Now()
290         cm.votedFor = cm.id
291         cm.dlog("becomes Candidate (currentTerm=%d); log=%v", savedCurrentTerm, cm.log)
292
293         votesReceived := 1
294
295         // Send RequestVote RPCs to all other servers concurrently.
296         for _, peerId := range cm.peerIds {
297             go func(peerId int) {
298                 args := RequestVoteArgs{
299                     Term:         savedCurrentTerm,
300                     CandidateId: cm.id,
301                 }
302                 var reply RequestVoteReply
303
304                 cm.dlog("sending RequestVote to %d: %v", peerId, args)
305                 if err := cm.server.Call(peerId, "ConsensusModule.RequestVote", args,
306                     cm.mu.Lock(),
307                     defer cm.mu.Unlock(),
308                     cm.dlog("received RequestVoteReply %v", reply))
309
310                 if cm.state != Candidate {
311                     cm.dlog("while waiting for reply, state = %v", cm.state)
312                     return
313                 }
314
315                 if reply.Term > savedCurrentTerm {
316                     cm.dlog("term out of date in RequestVoteReply")
317                     cm.becomeFollower(reply.Term)
318                     return
319                 } else if reply.Term == savedCurrentTerm {
320                     if reply.VoteGranted {
321                         votesReceived += 1
322                         if votesReceived*2 > len(cm.peerIds)+1 {
323                             // Won the election!
324                             cm.dlog("wins election with %d votes"
325                                 cm.startLeader()
326                                 return
327                             }

```

```
328                                     }
329                                 }
330                            }
331                        }(peerId)
332                    }
333
334                    // Run another election timer, in case this election is not successful.
335                    go cm.runElectionTimer()
336                }
337
338                // becomeFollower makes cm a follower and resets its state.
339                // Expects cm.mu to be locked.
340                func (cm *ConsensusModule) becomeFollower(term int) {
341                    cm.dlog("becomes Follower with term=%d; log=%v", term, cm.log)
342                    cm.state = Follower
343                    cm.currentTerm = term
344                    cm.votedFor = -1
345                    cm.electionResetEvent = time.Now()
346
347                    go cm.runElectionTimer()
348                }
349
350                // startLeader switches cm into a leader state and begins process of heartbeats.
351                // Expects cm.mu to be locked.
352                func (cm *ConsensusModule) startLeader() {
353                    cm.state = Leader
354                    cm.dlog("becomes Leader; term=%d, log=%v", cm.currentTerm, cm.log)
355
356                    go func() {
357                        ticker := time.NewTicker(3000 * time.Millisecond)
358                        defer ticker.Stop()
359
360                        // Send periodic heartbeats, as long as still leader.
361                        for {
362                            cm.leaderSendHeartbeats()
363                            <-ticker.C
364
365                            cm.mu.Lock()
366                            if cm.state != Leader {
367                                cm.mu.Unlock()
368                                return
369                            }
370                            cm.mu.Unlock()
371                        }
372                    }()
373                }
374
375                // leaderSendHeartbeats sends a round of heartbeats to all peers, collects their
376                // replies and adjusts cm's state.
```



```
377 func (cm *ConsensusModule) leaderSendHeartbeats() {
378     cm.mu.Lock()
379     if cm.state != Leader {
380         cm.mu.Unlock()
381         return
382     }
383     savedCurrentTerm := cm.currentTerm
384     cm.mu.Unlock()
385
386     for _, peerId := range cm.peerIds {
387         args := AppendEntriesArgs{
388             Term:    savedCurrentTerm,
389             LeaderId: cm.id,
390         }
391         go func(peerId int) {
392             cm.dlog("sending AppendEntries to %v: ni=%d, args=%+v", peerId, 0, args)
393             var reply AppendEntriesReply
394             if err := cm.server.Call(peerId, "ConsensusModule.AppendEntries", args, &reply); err != nil {
395                 cm.mu.Lock()
396                 defer cm.mu.Unlock()
397                 if reply.Term > savedCurrentTerm {
398                     cm.dlog("term out of date in heartbeat reply")
399                     cm.becomeFollower(reply.Term)
400                     return
401                 }
402             }
403         }(peerId)
404     }
405 }
```