

File - /Users/chadanlo/go/src/prr-labo2/labo2/mutex/mutex.go

```
1 /*
2 -----
3 Lab      : 02
4 File     : mutex.go
5 Authors  : François Burgener – Tiago P. Quinteiro
6 Date     : 03.12.2019
7
8 Goal     : Implements a Mutex in the algorithm of Carvalho et Roucairol
9 -----
10 */
11
12 package mutex
13
14 import (
15     "log"
16 )
17
18 /**
19  * ENUM declaration of the states
20  */
21 const (
22     REST = iota
23     WAITING
24     CRITICAL
25 )
26
27 /**
28  * Interface wanted for the Network
29  */
30 type Network interface {
31     REQ(stamp uint32, id uint16)
32     OK(stamp uint32, id uint16)
33     UPDATE(value uint32)
34 }
35
36 /**
37  * Passing stamp and id through channels
38  */
39 type Message struct {
40     stamp uint32
41     id uint16
42 }
43
44 /**
45  * Hides the values used by the mutex to handle his internal state
46  */
47 type mutexPrivate struct {
48     N uint16           // Total number of processes
49     me uint16          // The id of the Process
50     stamp uint32        // The logic clock
51     state uint8         // Rest, Waiting or CS
52     stampAsk uint32     // Stamp of the submitted request
53     pDiff map[uint16]bool // set of the processes we differed the OK
54     pWait map[uint16]bool // set of the processes we must wait a permission
55     netWorker Network
56 }
57
58 /**
59  * Hides the communication channels used by the mutex
60  */
61 type mutexChans struct {
62     reqChan chan Message
63     okChan  chan Message
64     endChan chan bool
65     updateChan chan uint32
66     askChan  chan bool
67     waitChan chan bool
68     resourceChan chan uint32
69 }
70
71 /**
72  * This is the class you may want to export
73  */
74 type Mutex struct {
75     private mutexPrivate
76     channels mutexChans
77     resource uint32
78     Debug bool
79 }
80
81 /**
82  * Constructor
83  * This method is responsible to initialize everything in order.
84  * ALWAYS CALL IT BEFORE DOING ANYTHING ELSE
85  */
86 func (m *Mutex) Init(id uint16, initialStamp uint32, numberOfProcess uint16, netWorker Network) {
87
88     m.private = mutexPrivate{
89         N:      numberOfProcess,
90         me:      id,
91         stamp:   initialStamp,
```

File - /Users/chadanlo/go/src/prr-labo2/labo2/mutex/mutex.go

```
92     state:    REST,
93     stampAsk: 0,
94     pDiff:    make(map[uint16]bool),
95     pWait:    make(map[uint16]bool),
96     netWorker: netWorker,
97 }
98
99 m.channels = mutexChans{
100     reqChan:    make(chan Message),
101     okChan:     make(chan Message),
102     endChan:    make(chan bool),
103     updateChan: make(chan uint32),
104     askChan:    make(chan bool),
105     waitChan:   make(chan bool),
106     resourceChan: make(chan uint32),
107 }
108
109 m.resource = 1000
110
111 // We start with some tokens already
112 m.initpWait()
113
114 // Here the manager starts
115 go m.manager()
116 }
117
118 // CLIENT SIDE API METHODS -----
119 // 1. Use Ask to start asking for the SC. Non blocking
120 // 2. Use Wait to wait over SC. Blocking
121 // 3. Use Update to set the value once you have SC
122 // 4. Use End to release SC
123 // Alternatively you can use GetResource to read the value when ever you want
124
125 /**
126  * Call this to ask the network for a future usage of the SC
127  */
128 func (m *Mutex) Ask() {
129     m.channels.askChan <- true
130 }
131
132 /**
133  * Block until the SC is ready
134  */
135 func (m *Mutex) Wait() {
136     <- m.channels.waitChan
137 }
138
139 /**
140  * Release the SC
141  */
142 func (m *Mutex) End() {
143     m.channels.endChan <- true
144 }
145
146 /**
147  * GETTER
148  */
149 func (m *Mutex) GetResource() uint32 {
150     m.channels.resourceChan <- 0
151     return <-m.channels.resourceChan
152 }
153
154 // SEVER SIDE API METHODS -----
155 // Use Req to notify incoming requests
156 // Use Ok to notify incoming Ok requests
157 // Use Update to set the value in SC
158
159 /**
160  * Pass an incoming network REQ here
161  */
162 func (m *Mutex) Req(stamp uint32, id uint16) {
163     message := Message{
164         stamp: stamp,
165         id:    id,
166     }
167     m.channels.reqChan <- message
168 }
169
170 /**
171  * Pass an incoming network OK here
172  */
173 func (m *Mutex) Ok(stamp uint32, id uint16) {
174     message := Message{
175         stamp: stamp,
176         id:    id,
177     }
178     m.channels.okChan <- message
179 }
180
181 /**
182  * SETTER: call this if you want to change the SC val
```

File - /Users/chadanlo/go/src/prr-labo2/labo2/mutex/mutex.go

```
183 * Never call it without being in SC (ask, wait, update, end)
184 */
185 func (m *Mutex) Update(value uint32) {
186     m.channels.updateChan <- value
187 }
188
189 // PRIVATE methods -----
190
191 /**
192  * This function runs in a goroutine
193  * It is the main handler of the mutex
194  * Every method called passes through here
195  */
196 func (m *Mutex) manager() {
197     log.Println("Stamp", m.private.stamp)
198
199     for {
200         select {
201             // ASK: Client called Ask()
202             case <- m.channels.askChan:
203                 m.handleAsk()
204
205             // END: Client released SC
206             case <- m.channels.endChan:
207                 m.handleEnd()
208
209             // P asked a token
210             case message := <- m.channels.reqChan:
211                 m.handleReq(message)
212
213             // P sent Ok
214             case message := <- m.channels.okChan:
215                 m.handleOk(message)
216
217             // Network told us to update, SETTER
218             case val := <- m.channels.updateChan:
219                 m.handleUpdate(val)
220
221             // Client asked value, GETTER
222             case <- m.channels.resourceChan:
223                 m.channels.resourceChan <- m.resource
224
225             default:
226                 // If we need to enter CS and don't wait on anyone
227                 if m.private.state == WAITING && len(m.private.pWait) == 0 {
228                     m.private.state = CRITICAL
229                     m.channels.waitChan <- true // we release our client
230                 }
231             }
232         }
233     }
234
235 /**
236  * Prepare the requests to other P
237  */
238 func (m *Mutex) handleAsk() {
239     if m.private.state == REST {
240         m.incrementClock(0)
241         m.private.state = WAITING
242         m.private.stampAsk = m.private.stamp
243         m.reqAll() // Sending req to the Ps to ask token
244     }
245
246     if m.Debug {
247         log.Printf("Mutex %d: Client asked me the CS", m.private.stamp)
248         log.Println("Stamp", m.private.stamp)
249     }
250 }
251
252 /**
253  * Releases the CS and sends Ok to differed P
254  */
255 func (m *Mutex) handleEnd() {
256     m.incrementClock(0)
257     m.private.state = REST // Leaving SC
258     m.private.netWorker.UPDATE(m.resource)
259     m.okAll() // Sending ok to the differed Ps
260
261     if m.Debug {
262         log.Printf("Mutex %d: Client released the CS", m.private.stamp)
263         log.Println("Stamp", m.private.stamp)
264     }
265 }
266
267 /**
268  * Handles incoming requests from other P
269  */
270 func (m *Mutex) handleReq(message Message) {
271     m.incrementClock(message.stamp) // Increment, max between mine and P
272
273     if m.private.state == REST {
```

File - /Users/chadanlo/go/src/prr-labo2/labo2/mutex/mutex.go

```
274
275     m.private.netWorker.OK(m.private.stamp, message.id) //Sending the signal
276     m.private.pWait[message.id] = true // Adding to waiting set
277 } else {
278     if m.private.state == CRITICAL ||
279        m.private.stampAsk < message.stamp ||
280        (m.private.stampAsk == message.stamp && m.private.me < message.id) {
281
282         m.private.pDiff[message.id] = true // We have to differ the obtain from other P
283     } else {
284         m.private.netWorker.OK(m.private.stamp, message.id) // Sending the signal
285         m.private.pWait[message.id] = true // Adding to waiting set
286         m.private.netWorker.REQ(m.private.stampAsk, message.id) // We send back the request
287     }
288 }
289
290 if m.Debug {
291     log.Printf("Mutex %d: Req received from %d", m.private.stamp, message.id)
292
293     for key, _ := range m.private.pWait {
294         log.Printf("Mutex: Waiting on him %d\n", key)
295     }
296     log.Println("Stamp", m.private.stamp)
297 }
298 }
299
300 /**
301  * Handles incoming Ok from other P
302  */
303 func (m *Mutex) handleOk(message Message) {
304     m.incrementClock(message.stamp) // Increment, max between mine and P
305     delete(m.private.pWait, message.id) // removing wait from here
306
307     if m.Debug {
308         log.Printf("Mutex %d: Ok received from %d", m.private.stamp, message.id)
309         log.Println("Stamp", m.private.stamp)
310     }
311 }
312
313 /**
314  * Handles incoming update (local or distant)
315  */
316 func (m *Mutex) handleUpdate(val uint32) {
317     log.Printf("Mutex %d: someone wants to update %d -> %d", m.private.stamp, m.resource, val)
318
319     m.resource = val
320
321     if m.Debug {
322         log.Println("Stamp", m.private.stamp)
323     }
324 }
325
326 /**
327  * Sends ok to all differed P in network
328  */
329 func (m *Mutex) okAll() {
330     for key, _ := range m.private.pDiff {
331         // Since we are sending ok, we now have to wait on him
332         m.private.pWait[key] = true
333         m.private.netWorker.OK(m.private.stamp, key)
334     }
335
336     // Clean the structure
337     m.private.pDiff = make(map[uint16]bool)
338 }
339
340 /**
341  * Sends req to all P in network you're waiting
342  */
343 func (m *Mutex) reqAll() {
344     for key, _ := range m.private.pWait {
345         m.private.netWorker.REQ(m.private.stamp, key)
346     }
347 }
348
349 /**
350  * Takes max and increments the stamp
351  * value uint32 - the value of the other stamp
352  */
353 func (m *Mutex) incrementClock(value uint32){
354     if value > m.private.stamp {
355         m.private.stamp = value
356     }
357
358     m.private.stamp += 1
359 }
360
361 /**
362  * Initialize the tokens this P has over the others
363  */
364 func (m *Mutex) initpWait(){
```

File - /Users/chadanlo/go/src/prr-labo2/labo2/mutex/mutex.go

```
365     for i := m.private.me + 1; i < m.private.N; i++ {  
366         m.private.pWait[i] = true  
367     }  
368 }
```