```go
/*
 -----------------------------------------------------------------------------------
 Lab         : 02
 File        : mutex.go
 Authors     : François Burgener - Tiago P. Quinteiro
 Date        : 03.12.2019

 Goal        : Implements a Mutex in the algorithm of Carvalho et Roucairol
 -----------------------------------------------------------------------------------
*/

package mutex

import (
    "log"
)

/**
 * ENUM declaration of the states
 */
const (
    REST = iota
    WAITING
    CRITICAL
)

/**
 * Interface wanted for the Network
 */
type Network interface {
    REQ(stamp uint32, id uint16)
    OK(stamp uint32, id uint16)
    UPDATE(value uint32)
}

/**
 * Passing stamp and id through channels
 */
type Message struct {
    stamp uint32
    id uint16
}

/**
 * Hides the values used by the mutex to handle his internal state
 */
type mutexPrivate struct {
    N uint16                // Total number of processes
    me uint16               // The id of the Process
    stamp uint32            // The logic clock
    state uint8         // Rest, Waiting or CS
    stampAsk uint32     // Stamp of the submitted request
    pDiff map[uint16]bool   // set of the processes we differed the OK
    pWait map[uint16]bool   // set of the processes we must wait a permission
    netWorker Network
}

/**
 * Hides the communication channels used by the mutex
 */
type mutexChans struct {
    reqChan      chan Message
    okChan       chan Message
    endChan      chan bool
    updateChan   chan uint32
    askChan      chan bool
    waitChan     chan bool
    resourceChan chan uint32
}

/**
 * This is the class you may want to export
 */
type Mutex struct {
    private  mutexPrivate
    channels mutexChans
    resource uint32
    Debug bool
}

/**
 * Constructor
 * This method is responsible to initialize everything in order.
 * ALWAYS CALL IT BEFORE DOING ANYTHING ELSE
 */
func (m *Mutex) Init(id uint16, initialStamp uint32, numberOfProcess uint16, netWorker Network) {

    m.private = mutexPrivate{
        N:         numberOfProcess,
        me:        id,
        stamp:     initialStamp,
        state:     REST,
        stampAsk:  0,
        pDiff:     make(map[uint16]bool),
        pWait:     make(map[uint16]bool),
        netWorker: netWorker,
    }

    m.channels = mutexChans{
        reqChan:      make(chan Message),
        okChan:       make(chan Message),
        endChan:      make(chan bool),
        updateChan:   make(chan uint32),
        askChan:      make(chan bool),
        waitChan:     make(chan bool),
        resourceChan: make(chan uint32),
    }

    m.resource = 1000

    // We start with some tokens already
    m.initpWait()

    // Here the manager starts
    go m.manager()
}

// CLIENT SIDE API METHODS --------------------------------
// 1. Use Ask to start asking for the SC. Non blocking
// 2. Use Wait to wait over SC. Blocking
// 3. Use Update to set the value once you have SC
// 4. Use End to release SC
// Alternatively you can use GetResource to read the value when ever you want

/**
 * Call this to ask the network for a future usage of the SC
 */
func (m *Mutex) Ask() {
    m.channels.askChan <- true
}

/**
 * Block until the SC is ready
 */
func (m *Mutex) Wait() {
    <- m.channels.waitChan
}

/**
 * Release the SC
 */
func (m *Mutex) End() {
    m.channels.endChan <- true
}

/**
 * GETTER
 */
func (m *Mutex) GetResource() uint32 {
    m.channels.resourceChan <- 0
    return <-m.channels.resourceChan
}

// SEVER SIDE API METHODS ---------------------------------
// Use Req to notify incoming requests
// Use Ok to notify incoming Ok requests
// Use Update to set the value in SC

/**
 * Pass an incoming network REQ here
 */
func (m *Mutex) Req(stamp uint32, id uint16) {
    message := Message{
        stamp: stamp,
        id:    id,
    }
    m.channels.reqChan <- message
}

/**
 * Pass an incoming network OK here
 */
func (m *Mutex) Ok(stamp uint32, id uint16) {
    message := Message{
        stamp: stamp,
        id:    id,
    }
    m.channels.okChan <- message
}

/**
 * SETTER: call this if you want to change the SC val
 * Never call it without being in SC (ask, wait, update, end)
 */
func (m *Mutex) Update(value uint32) {
	m.channels.updateChan <- value
}

// PRIVATE methods -----------------------------------

/**
 * This function runs in a goroutine
 * It is the main handler of the mutex
 * Every method called passes through here
 */
func (m *Mutex) manager() {
	log.Println("Stamp", m.private.stamp)

	for {
		select {
		// ASK: Client called Ask()
		case <- m.channels.askChan:
			m.handleAsk()

		// END: Client released SC
		case <- m.channels.endChan:
			m.handleEnd()

		// P asked a token
		case message := <- m.channels.reqChan:
			m.handleReq(message)

		// P sent Ok
		case message:= <- m.channels.okChan:
			m.handleOk(message)

		// Network told us to update, SETTER
		case val := <- m.channels.updateChan:
			m.handleUpdate(val)

		// Client asked value, GETTER
		case <- m.channels.resourceChan:
			m.channels.resourceChan <- m.resource

		default:
			// If we need to enter CS and don't wait on anyone
			if m.private.state == WAITING && len(m.private.pWait) == 0 {
				m.private.state = CRITICAL
				m.channels.waitChan <- true // we release our client
			}
		}
	}
}

/**
 * Prepare the requests to other P
 */
func (m *Mutex) handleAsk() {
	if m.private.state == REST {
		m.incrementClock(0)
		m.private.state = WAITING
		m.private.stampAsk = m.private.stamp
		m.reqAll() // Sending req to the Ps to ask token
	}

	if m.Debug {
		log.Printf("Mutex %d: Client asked me the CS", m.private.stamp)
		log.Println("Stamp", m.private.stamp)
	}
}

/**
 * Releases the CS and sends Ok to differed P
 */
func (m *Mutex) handleEnd() {
	m.incrementClock(0)
	m.private.state = REST      // Leaving SC
	m.private.netWorker.UPDATE(m.resource)
	m.okAll() // Sending ok to the differed Ps

	if m.Debug {
		log.Printf("Mutex %d: Client released the CS", m.private.stamp)
		log.Println("Stamp", m.private.stamp)
	}
}

/**
 * Handles incoming requests from other P
 */
func (m *Mutex) handleReq(message Message) {
	m.incrementClock(message.stamp) // Increment, max between mine and P

	if m.private.state == REST {

        m.private.netWorker.OK(m.private.stamp, message.id) //Sending the signal
        m.private.pWait[message.id] = true // Adding to waiting set
    } else {
        if m.private.state == CRITICAL ||
            m.private.stampAsk < message.stamp ||
            (m.private.stampAsk == message.stamp && m.private.me < message.id) {

            m.private.pDiff[message.id] = true // We have to differ the obtain from other P
        } else {
            m.private.netWorker.OK(m.private.stamp, message.id) // Sending the signal
            m.private.pWait[message.id] = true // Adding to waiting set
            m.private.netWorker.REQ(m.private.stampAsk, message.id) // We send back the request
        }
    }

    if m.Debug {
        log.Printf("Mutex %d: Req received from %d", m.private.stamp, message.id)

        for key, _ := range m.private.pWait  {
            log.Printf("Mutex: Waiting on him %d\n", key)
        }
        log.Println("Stamp", m.private.stamp)
    }
}

/**
 * Handles incoming Ok from other P
 */
func (m *Mutex) handleOk(message Message) {
    m.incrementClock(message.stamp) // Increment, max between mine and P
    delete(m.private.pWait, message.id) // removing wait from here

    if m.Debug {
        log.Printf("Mutex %d: Ok received from %d", m.private.stamp, message.id)
        log.Println("Stamp", m.private.stamp)
    }
}

/**
 * Handles incoming update (local or distant)
 */
func (m *Mutex) handleUpdate(val uint32) {
    log.Printf("Mutex %d: someone wants to update %d -> %d", m.private.stamp, m.resource, val)

    m.resource = val

    if m.Debug {
        log.Println("Stamp", m.private.stamp)
    }
}

/**
 * Sends ok to all differed P in network
 */
func (m *Mutex) okAll() {
    for key, _ := range m.private.pDiff {
        // Since we are sending ok, we now have to wait on him
        m.private.pWait[key] = true
        m.private.netWorker.OK(m.private.stamp, key)
    }

    // Clean the structure
    m.private.pDiff =  make(map[uint16]bool)
}

/**
 * Sends req to all P in network you're waiting
 */
func (m *Mutex) reqAll() {
    for key, _ := range m.private.pWait  {
        m.private.netWorker.REQ(m.private.stamp, key)
    }
}

/**
 * Takes max and increments the stamp
 * value uint32 -  the value of the other stamp
 */
func (m *Mutex) incrementClock(value uint32){
    if value > m.private.stamp {
        m.private.stamp = value
    }

    m.private.stamp += 1
}

/**
 * Initialize the tokens this P has over the others
 */
func (m *Mutex) initpWait(){
    for i := m.private.me + 1; i < m.private.N; i++ {
        m.private.pWait[i] = true
    }
}
```