```go
/*
 -----------------------------------------------------------------------------------
 Lab         : 03
 File        : manager.go
 Authors     : François Burgener - Tiago P. Quinteiro
 Date        : 10.12.19

 Goal        : Implements the manager for the bully algorithm of Chang and Roberts
 -----------------------------------------------------------------------------------
*/
package manager

import (
	"log"
)

/**
 * ENUM declaration of the states
 */
const (
	NOTIFICATION = iota
	RESULT
)

/**
 * Interface wanted for the Network
 */
type Network interface {
	EmitNotif(map[uint16]uint16)
	EmitResult(uint16, map[uint16]bool)
}

/**
 * private utility struct
 * to send through channels
 * a result message
 */
type resultMessage struct {
	id uint16
	visitedResult map[uint16]bool
}

/**
 * Manager class
 */
type Manager struct {
	N uint16
	me uint16
	aptitude uint16
	state uint8
	elected uint16
	asked bool
	debug bool
	network Network
	chanAskElection chan bool
	chanGiveElection chan uint16
	chanNotification chan map[uint16]uint16
	chanResult chan resultMessage
	chanAsk chan bool
}

/**
 * Constructor
 * @param N number of Processes
 * @param me id of this Process
 * @param aptitude the aptitude of this Process
 * @param network a struct which represents the network layer
 */
func (m *Manager) Init(N uint16, me uint16, aptitude uint16, network Network) {
	log.Println("Manager : Initialization of the manager")
	m.N = N
	m.me = me
	m.aptitude = aptitude
	m.network = network
	m.state = RESULT
	m.asked = false

	//Channels
	m.chanAskElection = make(chan bool)
	m.chanGiveElection = make(chan uint16)
	m.chanNotification = make(chan map[uint16]uint16)
	m.chanResult = make(chan resultMessage)
	m.chanAsk = make(chan bool)

	// Debug
	m.debug = true

	go m.handler()
}

/**
```

```go
 * Once Init, this handler will treat incoming requests
 * from Task and Network
 */
func (m *Manager) handler() {
	for {
		select {
		case <- m.chanAskElection:
			m.handleElection()
		case notifMap := <- m.chanNotification:
			m.handleNotification(notifMap)
		case resultMessage := <- m.chanResult:
			m.handleResult(resultMessage)
		case m.asked = <- m.chanAsk:
		default:
			if m.state == RESULT && m.asked {
				if m.debug {
					log.Println("Manager : Send elected processus")
				}
				m.asked = false
				m.chanGiveElection <- m.elected
			}
		}
	}
}

// API for network

/**
 * Submits a Notification message to manager from network
 */
func (m *Manager) SubmitNotification(notifMap map[uint16]uint16) {
	m.chanNotification <- notifMap
}

/**
 * Submits a result message to manager from network
 */
func (m *Manager) SubmitResult(id uint16, resultMap map[uint16]bool) {
	m.chanResult <- resultMessage{
		id:            id,
		visitedResult: resultMap,
	}
}

// API for Task

/**
 * Tells manager to start an election
 */
func (m *Manager) RunElection() {
	m.chanAskElection <- true
}

/**
 * Get the elected id
 */
func (m *Manager) GetElected() uint16 {
	m.chanAsk <- true
	return <- m.chanGiveElection
}

// Privates

/**
 * Runs an election
 */
func (m *Manager) handleElection() {
	l := m.createNewMap()
	m.sendNotification(l)
}

/**
 * Handles a Notification request
 * @param notifMap map of id and aptitudes
 */
func (m *Manager) handleNotification(notifMap map[uint16]uint16) {
	if m.debug {
		log.Println("Manager : Received NOTIFICATION ")
	}

	_, isInside := notifMap[m.me] // Test if I'm here
	if isInside {
		m.elected = findMax(notifMap)
		m.sendResult()
	} else {
		notifMap[m.me] = m.aptitude // Add myself in map
		m.sendNotification(notifMap)
	}
}

/**
```

```go
 * Handles a Result request
 * @param resultMessage
 */
func (m *Manager) handleResult(resultMessage resultMessage) {
    if m.debug {
        log.Println("Manager : Received RESULT, new boss is ", resultMessage.id)
    }

    i := resultMessage.id
    resultMap := resultMessage.visitedResult

    _, isInside := resultMap[m.me] // Test if I'm here
    if isInside {
        // Nothing to do ¯\_(ツ)_/¯
    } else if m.state == RESULT && m.elected != i {
        l := m.createNewMap()

        m.sendNotification(l)
    } else if m.state == NOTIFICATION {
        m.elected = i
        m.sendResult()
    }
}

/**
 * Calls network and emit notification
 * @param map of ids and aptitudes
 */
func (m *Manager) sendNotification(_map map[uint16]uint16) {
    m.network.EmitNotif(_map)
    m.state = NOTIFICATION
}

/**
 * Calls network and emit result
 */
func (m *Manager) sendResult() {
    resultMap := make(map[uint16]bool)
    resultMap[m.me] = true

    m.network.EmitResult(m.elected,resultMap)
    m.state = RESULT
}

/**
 *
 */
func (m *Manager) createNewMap() map[uint16]uint16 {
    l := make(map[uint16]uint16)
    l[m.me] = m.aptitude
    return l
}

/**
 * Utility function to find max
 * @param m Map where you want to find max
 */
func findMax (m map[uint16]uint16) uint16 {
    var id, max uint16 = 0, 0

    for key, val := range m {
        if val > max {
            max = val
            id = key
        }
    }

    return id
}
```