XMP 2013 Assignment 2 (Programming)

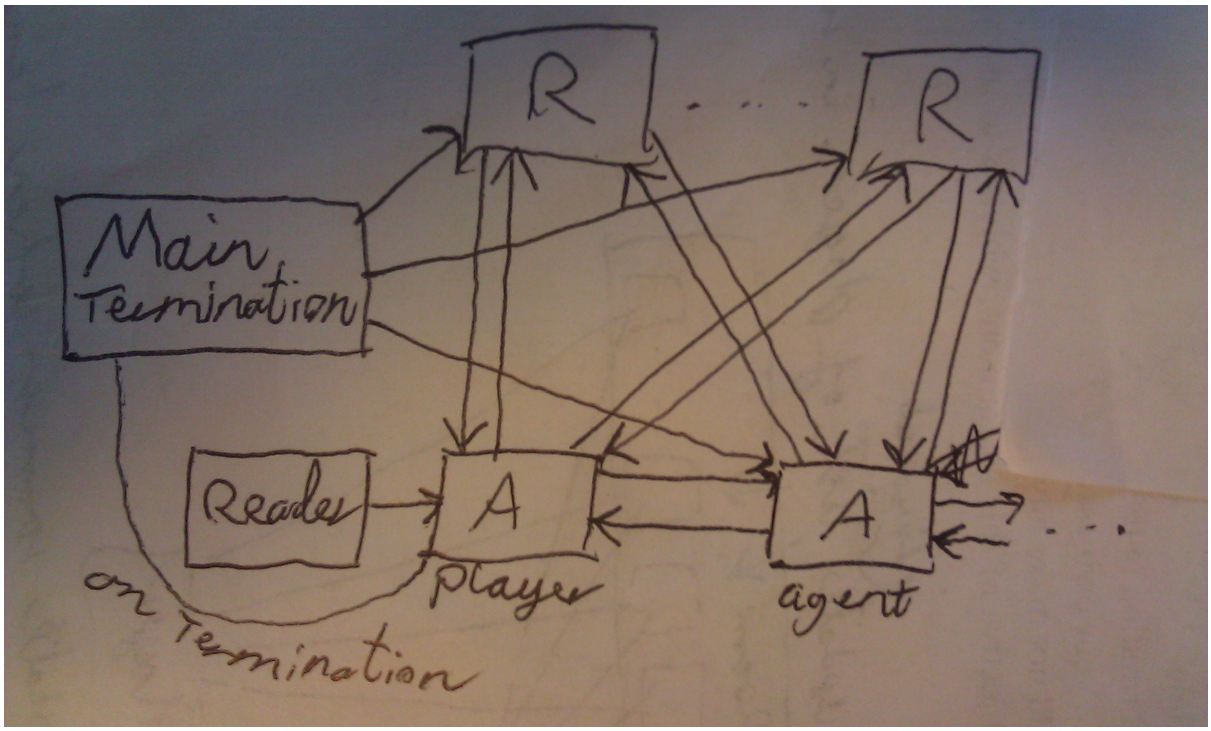
Henrik Bendt (gwk553)

2. januar 2014

1 Design

The program is written in Go (golang). Some limitations in Go, which affects this implementation, is for example when waiting on input from multiple channels, one must loop over all channels and skip if there is no message on the channel, which makes it a busy wait (somewhat expensive). One can close channels, but I found it easier to manage termination with designated quit channels to each subprocess. A channel is strictly typed, why I have made a struct to handle multi field channels. Also, a channel is bidirectional, but this is dangerous to use (and not very RPC), why this implementation only uses a channel in one direction.

All actors (players and agent/AI) and rooms use unique bichannels (a channel in each direction). An actor have a list of unique bichannels to rooms and to actors and a room only has a list of unique bichannels to actors. Each room and actor has an id, which serves as index to these lists and is how rooms communicates to actors, what rooms and actors to contact during a certain action.



Figur 1: Figure of the channel network utilized in this implementation. The top boxes are rooms, which and the bottum are actors (the first is the player). All channels have directions. On termination, the player moves to the main termination method, where only quit-channels to each other process exists. I have not named the channels as I do not know what they should be called (it is the sender/reciever that defines these channels).

When entering and leaving a room, the actor must tell all other actors in the room, that it is entering or leaving. This must not block the room nor the actors. One way to handle this is to use the room to message all actors the news, and to make the channels buffered (with size equal to number of actors). As this implementation is very strict on recieved messages from rooms to actors (it must be a response to the previous request, like look), this is not possible (as it would not recognize the use of the message). Instead of adapting to the above (which is possible in this implementation), this implementation makes the entering/leaving actor tell all other actors the news. Thus, when an actor is leaving, it tells the room, gets the ids (indexes) of the actors in the room and then messages these actors of the news. This should always take a finite time, as each actor checks all these input channels before every act, but to make the implementation more smooth and so an actor does not wait on another actor, the channels are 2-buffered (to handle if for example timeslices between the actor threads are very uneven).

To make sure the player does not block the above events, a thread is forked to handle input reads by the user, and send these on a channel to the user.

Each actor has a unique channel to each room to communicate by. This channel, though less RPC, communicates with messages (a custom struct), where keywords and arguments are stored. Another, more RPC, way to do this could be to make individual channels for individual purposes between each actor and room, like a channel for entering/leaving, taking objects, leaving objects, etc. This however is very extensive and a great deal of work.

Rooms are simply states of what is currently hold in the room and only responds on requests from actors, never act on its own.

Objects/items are kept as strings and send via messages over channels, when taking/leaving them.

1.1 Initialisation

Rooms are the first to be initialised, then agents and then the master process goes to the player method.

Rooms are initialised by a pseudo random strategy, where a random number of doors are created for each room. This random number is set to be at least 1 (unless there are no more rooms to be made), and it favors rooms in direction of order N, S, E, W. The initialization of rooms was simplified to make it easier to implement.

Note that this strategy is recursive and will have a chance of creating mazes of rooms, where some rooms overlap.

Objects are initially given to rooms by an even distribution. They are hardcoded to be a number of gold.

In main, one can edit the number of rooms, the number of Agents (AI) and the number of objects.

1.2 Termination

Player is the only one who can shut down the game. Player initiates termination by first leaving the room (not entering another room) and then clearing the inboxes of the channels from all other agents. It now assumes that it will be contacted no more (or at least there is enough space on each channel for another player to contact it twice, which should not happen when the player is in no room). The reader process exits when reading the exit command.

Each unique quit-channel for each agent is then called (by the main process), forcing all agents to exit the same way as the player. All agents must exit before any room exits so an agent don't interact with a dead room. When all agents are dead, the rooms are also told to exit by calling each room on its unique quit-channel. As all agents are dead at this point, it is safe for the rooms to exit without further ado.

The program terminates somewhat slow, because each agent sleeps 1 second at a time (before acting), and thus the termination can take $numAgents + numAgents * timeForAnAction$ seconds to terminate.

2 Source Code

```
package main

import (
    "fmt"
    "math/rand"
    "time"
    "container/list"
    "strconv"
)

/*
    2013-01-02, Henrik Bendt, MUD game
    Creates some rooms and agents and objects.
    Player can move between rooms and interact with objects.
*/

//Struct for messages send over channels
type Message struct {
    Command string
    Object []string //first object used in case of take action
    NumActors int
    RoomIndex int //door
    ActorIndexes []int //enter/leave
}
//Struct for a bidirectional channel, where a process only sends on one and recieves on another.
//Naming is a bit messed up because this is also used for channels between actors.
type TupleMesCh struct {
    ActorRoom chan Message
    RoomActor chan Message
}

/*
    The process for a room.
    doors is a list doors in the room, contianing indexes to the rooms with the doors.
    A room has a list of channels to all actors and a quit channel.
*/
func room(doors []int, actorRoomChs []TupleMesCh, quit chan int){
    objects := list.New()
    actors := list.New()

    for {
        //check on all actor channels until told to exit
        for i, c := range actorRoomChs {
            select {
            case <-quit: //die
                return
            case msg :=<- c.ActorRoom:
                switch msg.Command {
                case "enter":
                    //notify all other in room
                    is := make([]int, actors.Len())
                    j := 0
                    for e := actors.Front(); e != nil; e = e.Next() {
                        is[j] = e.Value.(int)
                        j++
                    }
```

```
        c.RoomActor<-Message{Command: "actorEntered", ActorIndexes: is[:]}
        actors.PushFront(i) // save index of actor
    case "moveOut":
        for e := actors.Front(); e != nil; e = e.Next() {
            if e.Value.(int) == i{ // actor is leaving
                actors.Remove(e)
                break
            }
        }
        is := make([]int, actors.Len())
        j := 0
        for e := actors.Front(); e != nil; e = e.Next() {
            is[j] = e.Value.(int)
            j++
        }
        c.RoomActor<-Message{Command: "actorLeft", ActorIndexes: is[:]}
    case "take": //take object in room
        object := [1]string{""}
        if len(msg.Object) >= 1 {
            for e := objects.Front(); e != nil; e = e.Next() {
                if e.Value == msg.Object[0] {
                    objects.Remove(e)
                    object[0] = msg.Object[0]
                    //fmt.Println("Object", msg.Object, "taken.")
                    break
                }
            }
        }
        c.RoomActor<-Message{Object: object[:]}
    case "leave": //leave object in room
        if msg.Object[0] != "" {
            objects.PushFront(msg.Object[0])
            fmt.Println("object", msg.Object[0], "left in room.")
        }
    case "look": //list actors and objects
        objectList := make([]string, objects.Len())
        e := objects.Front()
        for i:= range objectList {
            objectList[i] = e.Value.(string)
            e = e.Next()
        }
        c.RoomActor<-Message{Command: "look", Object: objectList, NumActors: actors.Len(
    case "N": //returns index to room north if exists, else -1
        roomId := doors[0]
        c.RoomActor<-Message{Command: "N", RoomIndex: roomId}
    case "S": //returns index to room south if exists, else -1
        roomId := doors[1]
        c.RoomActor<-Message{Command: "S", RoomIndex: roomId}
    case "E": //returns index to room east if exists, else -1
        roomId := doors[2]
        c.RoomActor<-Message{Command: "E", RoomIndex: roomId}
    case "W": //returns index to room west if exists, else -1
        roomId := doors[3]
        c.RoomActor<-Message{Command: "W", RoomIndex: roomId}
    default: //should not happen
        fmt.Println("Something went wrong.")
    }
    default : //skip
```

```
            }
        }
    }
}

/*
    Ai can move and take/leave objects.
    Terminates when recieving message on quit channel
*/
func ai(roomChs, playerChs []TupleMesCh, quit chan int){
    currRoom := 0 //init index of current room
    object := ""

    enterRoom := func(){
        roomChs[currRoom].ActorRoom<-Message{Command: "enter"}
        msg :=<-roomChs[currRoom].RoomActor
        for i:=0;i<len(msg.ActorIndexes);i++ { //Inform the men!
            playerChs[msg.ActorIndexes[i]].ActorRoom<-Message{Command: "actorEntered"}
        }
    }
    leaveRoom := func(){
        roomChs[currRoom].ActorRoom<-Message{Command: "moveOut"}
        msg :=<-roomChs[currRoom].RoomActor
        for i:=0;i<len(msg.ActorIndexes);i++ { //Inform the men!
            playerChs[msg.ActorIndexes[i]].ActorRoom<-Message{Command: "actorLeft"}
        }
    }
    //init enter
    enterRoom()
    for {
        time.Sleep(1 * time.Second) //Make an action every second
        for _,c:= range playerChs { //check if actor entered/left room
            select {
                case msg := <- c.RoomActor:
                    switch msg.Command {
                    case "actorEntered":
                    case "actorLeft": //not sure what to use this info for.
                    }
                default: //skip
            }
        }
        select {
        case <-quit: //die
            //fmt.Println("AI quit")
            roomChs[currRoom].ActorRoom<-Message{Command: "moveOut"}
            <-roomChs[currRoom].RoomActor
            return
        default :
            switch rand.Intn(3) {
            case 0: //Move
                //fmt.Println("AI move")
                r := make([]int, 4)
                roomChs[currRoom].ActorRoom<-Message{Command: "N"}
                msg:=<-roomChs[currRoom].RoomActor
                r[0] = msg.RoomIndex
                roomChs[currRoom].ActorRoom<-Message{Command: "S"}
                msg=<-roomChs[currRoom].RoomActor
                r[1] = msg.RoomIndex
```

```
                roomChs[currRoom].ActorRoom<-Message{Command: "E"}
                msg=<-roomChs[currRoom].RoomActor
                r[2] = msg.RoomIndex
                roomChs[currRoom].ActorRoom<-Message{Command: "W"}
                msg=<-roomChs[currRoom].RoomActor
                r[3] = msg.RoomIndex
                flag := false //Check if any doors exists
                for _,v :=range r {
                    if v >= 0 {
                        flag = true
                    }
                }
                if flag {
                    for {
                        i := rand.Intn(4)
                        if r[i] >= 0 {
                            leaveRoom()
                            currRoom = r[i]
                            enterRoom()
                            break
                        }
                    }
                }
            case 1: //take object if any
                if object == "" {
                    roomChs[currRoom].ActorRoom<-Message{Command: "look"}
                    msg :=<-roomChs[currRoom].RoomActor
                    lenObj := len(msg.Object)
                    if lenObj > 0 {
                        objToTake := msg.Object[rand.Intn(lenObj)]
                        objList := [1]string{objToTake}
                        roomChs[currRoom].ActorRoom<-Message{Command: "take", Object: objList[:]}
                        msg :=<-roomChs[currRoom].RoomActor
                        if msg.Object[0] == objToTake {
                            object = msg.Object[0]
                        }
                    }
                }
            case 2: //leave object if any
                //fmt.Println("AI leave")
                if object != "" {
                    objList := [1]string{object}
                    roomChs[currRoom].ActorRoom<-Message{Command: "leave", Object: objList[:]}
                    object = ""
                }
            }
        }
    }
}


func formatRoomLookString(i int, direction string) string {
    response := "There is "
    if i < 0 {
        response += "no "
    } else {
        response += "a "
    }
```

```
    return response+"door to the "+direction
}

/*
    The player process reacts on input from stdin.
*/
func player(roomChs, playerChs []TupleMesCh){
    currRoom := 0 //index of current room
    object := ""
    var s string
    inCh := make(chan string)

    enterRoom := func(){
        roomChs[currRoom].ActorRoom<-Message{Command: "enter"}
        msg :=<-roomChs[currRoom].RoomActor
        for i:=0; i<len(msg.ActorIndexes); i++ { //Inform the men!
            playerChs[i].ActorRoom<-Message{Command: "actorEntered"}
        }
    }

    leaveRoom := func(){
        roomChs[currRoom].ActorRoom<-Message{Command: "moveOut"}
        msg :=<-roomChs[currRoom].RoomActor
        for i:=0;i<len(msg.ActorIndexes);i++ { //Inform the men!
            playerChs[msg.ActorIndexes[i]].ActorRoom<-Message{Command: "actorLeft"}
        }
    }

    moveRoom := func (roomIndex int, direction string) {
        if roomIndex < 0 {
            fmt.Println("There is no door to the "+direction)
        } else {
            leaveRoom()
            currRoom = roomIndex
            enterRoom()
        }
    }

    checkEnterLeave := func() {
        for i:=0;i<2;i++{ //this is just to potentially clear the inbox of channel (buffer=2)
            for i, c := range playerChs { //check if actor entered/left room
                select {
                    case msg := <- c.RoomActor:
                        switch msg.Command {
                        case "actorEntered":
                            fmt.Println("Actor", i, "entered")
                        case "actorLeft":
                            fmt.Println("Actor", i, "left")
                        }
                        //We now know an actor has entered room.
                    default: //skip
                }
            }
        }
    }

    playerInput := func(c chan string) {
        var s string
```

```
    //c := make(chan string)
    for {
        fmt.Scan(&s)
        c<-s
        if s=="exit" {
            return
        }
    }
}

//init enter
enterRoom()
go playerInput(inCh)

for {
    select {
    case s = <-inCh:
        switch s {
        case "exit": //quit
            fmt.Println("Exiting")
            //leave room, so other actors does not contact
            roomChs[currRoom].ActorRoom<-Message{Command: "moveOut"}
            <-roomChs[currRoom].RoomActor
            checkEnterLeave() //clear out channel one last time
            return
        case "look" :
            fmt.Println("Looking at room", currRoom)
            roomChs[currRoom].ActorRoom<-Message{Command: "look"}
            msg:=<-roomChs[currRoom].RoomActor
            fmt.Println("Objects in room:",msg.Object)
            fmt.Println("Players in room:",msg.NumActors)
            roomChs[currRoom].ActorRoom<-Message{Command: "N"}
            msg =<-roomChs[currRoom].RoomActor
            fmt.Println(formatRoomLookString(msg.RoomIndex, "North"))
            roomChs[currRoom].ActorRoom<-Message{Command: "S"}
            msg =<-roomChs[currRoom].RoomActor
            fmt.Println(formatRoomLookString(msg.RoomIndex, "South"))
            roomChs[currRoom].ActorRoom<-Message{Command: "E"}
            msg =<-roomChs[currRoom].RoomActor
            fmt.Println(formatRoomLookString(msg.RoomIndex, "East"))
            roomChs[currRoom].ActorRoom<-Message{Command: "W"}
            msg =<-roomChs[currRoom].RoomActor
            fmt.Println(formatRoomLookString(msg.RoomIndex, "West"))
            if object != "" {
                fmt.Println("You are holding object "+object)
            } else {
                fmt.Println("You are not holding any objects")
            }
        case "take" :
            if object != "" {
                fmt.Println("You already hold an object")
            } else {
                var obj string
                fmt.Scan(&obj) //TODO FIX
                objList := [1]string{obj}
                roomChs[currRoom].ActorRoom<-Message{Command: "take", Object: objList[:]}
                msg :=<-roomChs[currRoom].RoomActor
                if len(msg.Object) > 0 {
```

```
                        object = msg.Object[0]
                        fmt.Println("You took object", object, "from the room")
                    } else {
                        fmt.Println("No such object (", obj, ") found in the room")
                    }
                }
            case "leave" :
                if object == "" {
                    fmt.Println("You are not holding any objects")
                } else {
                    objList := [1]string{object}
                    roomChs[currRoom].ActorRoom<-Message{Command: "leave", Object: objList[:]}
                    fmt.Println("You left object", object, "in the room")
                    object = ""
                }
            case "N" :
                roomChs[currRoom].ActorRoom<-Message{Command: "N"}
                msg :=<-roomChs[currRoom].RoomActor
                moveRoom(msg.RoomIndex, "North")
            case "S" :
                roomChs[currRoom].ActorRoom<-Message{Command: "S"}
                msg :=<-roomChs[currRoom].RoomActor
                moveRoom(msg.RoomIndex, "South")
            case "E" :
                roomChs[currRoom].ActorRoom<-Message{Command: "E"}
                msg :=<-roomChs[currRoom].RoomActor
                moveRoom(msg.RoomIndex, "East")
            case "W" :
                roomChs[currRoom].ActorRoom<-Message{Command: "W"}
                msg :=<-roomChs[currRoom].RoomActor
                moveRoom(msg.RoomIndex, "West")
            default:
                fmt.Println("Unknown command")
            }

        default:
            time.Sleep(100 * time.Millisecond) //must exist to not spin crash

            checkEnterLeave()
        }
    }
}


/*
    Recursivly initialises and forks rooms.
    numRooms is the number of rooms left to be made
    direction is the direction of the previous room (reserved)
    index is the place of the current room
    masterIndex is the start index of rooms not initiated yet
    (can be used by this room for its doors)
*/
func newRoom(numRooms, direction, prevIndex, index, masterIndex int, actorChs [][]TupleMesCh) (int,
    rand.Seed(time.Now().UTC().UnixNano()) //seed random generator
    var numDoors int
    if direction < 0 {
        numDoors = rand.Intn(3)+1
        if numDoors > numRooms { //+1 to make it easier, could be rand.Intn(4)
```

```
            numDoors = numRooms
        }
    } else {
        numDoors = rand.Intn(2)+1 // There is already 1 door in use
        if numDoors > numRooms { //+1 to make it easier, could be rand.Intn(3)
            numDoors = numRooms
        }
    }
    newNumRooms := numRooms-numDoors
    doors := make([]int, 4)
    for i:=range doors {
        doors[i] = -1
    }
    quitChs := make([]chan int, 0)
    var newQuitChs []chan int
    newMasterIndex := masterIndex+numDoors

    if direction == 0 || direction == 2 {
        direction += 1
    } else {
        direction -= 1
    }
    if direction >= 0 {
        doors[direction] = prevIndex
    }

    for i:=0;i<numDoors;i++ {
        var a int
        if i == direction {
            a = numDoors-1
        } else { //previous room
            a = i
        }
        nextIndex :=  masterIndex+a
        doors[a] = nextIndex
        newNumRooms, newMasterIndex, newQuitChs = newRoom(newNumRooms, a, index, nextIndex, newMaste
        tmpQuitChs := make([]chan int, len(quitChs)+len(newQuitChs))
        copy(tmpQuitChs[0:len(newQuitChs)], newQuitChs)
        copy(tmpQuitChs[len(newQuitChs):], quitChs[:])
        quitChs = tmpQuitChs[:]
    }
    quitCh := make(chan int)
    roomChs := make([]TupleMesCh, len(actorChs[index]))
    copy(roomChs, actorChs[index])
    go room(doors, roomChs, quitCh)
    tmpQuitChs := make([]chan int, len(quitChs)+1)
    copy(tmpQuitChs[0:len(quitChs)], quitChs[:])
    tmpQuitChs[len(tmpQuitChs)-1] = quitCh
    quitChs = tmpQuitChs
    return newNumRooms, newMasterIndex, quitChs[:]
}


/*
    Setup the game.
*/
func setup(numRooms, numAIs, numPlayers, numObjects int) ([]chan int, []chan int){
    fmt.Println("setup")
```

```
fmt.Println("Number of rooms:", numRooms, "Number of AIs:", numAIs, "Number of objects:", numObj
numActors := numAIs+numPlayers
roomChs := make([][]TupleMesCh, numRooms)
//First make, for all rooms, channels to each player
for i:= range roomChs {
    roomChs[i] = make([]TupleMesCh, numActors)
    for j:= range roomChs[i] {
        roomChs[i][j] = TupleMesCh{make(chan Message), make(chan Message)}
    }
}

//Init rooms
_, _, quitRoomChs := newRoom(numRooms-1, -1, -1, 0, 1, roomChs)

//Distribute objects to rooms
for i:=0; i< numObjects; i++ {
    objList := [1]string{strconv.Itoa(i)+" gold"}
    roomChs[i%numRooms][0].ActorRoom<-Message{Command: "leave", Object: objList[:]}
}

//Now make all actor channels
actorChs := make([][]chan Message, numActors)
for i:= range actorChs {
    actorChs[i] = make([]chan Message, numActors)
    for j:= range actorChs[i] {
        actorChs[i][j] = make(chan Message, 2) //buffered channel
    }
}

 //make AI
quitAiChs := make([]chan int, numAIs)
for i:= 0; i<numAIs; i++ {
    //make actor-room channels
    actorRoomChs := make([]TupleMesCh, numRooms)
    for j:= range actorRoomChs{
        actorRoomChs[j] = roomChs[j][i]
    }

    //make actor-actor channels
    actorActorChs := make([]TupleMesCh, numActors)
    for j:= range actorActorChs {
        actorActorChs[j] = TupleMesCh{actorChs[i][j], actorChs[j][i]}
    }

    quitCh := make(chan int)
    quitAiChs[i] = quitCh
    go ai(actorRoomChs, actorActorChs, quitCh)
}


//goto player
playerRoomChs := make([]TupleMesCh, numRooms)
actorActorChs := make([]TupleMesCh, numActors)
for i := range playerRoomChs {
    playerRoomChs[i] = roomChs[i][numActors-1] //copy last channel set of each room to player
}
for i:= range actorActorChs {
    actorActorChs[i] = TupleMesCh{actorChs[numActors-1][i], actorChs[i][numActors-1]}
```

```
    }
    player(playerRoomChs, actorActorChs)
    return quitAiChs, quitRoomChs
}


func main() {
    numberOfRooms := 10
    numberOfAI := 10
    numberOfObjects := 5
    numberOfPlayers := 1 //Is assumed to be 1 for the time being
    quitAiChs, quitRoomChs := setup(numberOfRooms, numberOfAI, numberOfPlayers, numberOfObjects)
    //kill all agents/ai
    for _,c:= range quitAiChs {
        c<--1
        fmt.Println("ai exited")
    }
    //kill all rooms
    for _,c:= range quitRoomChs {
        c<--1
        fmt.Println("room exited")
    }

    fmt.Println("player exited")
}
```