Task finale
LABORATORIO DI SISTEMI SOFTWARE

Pecci Federica Varini Chiara

Dicembre 2018

1 Sprint 0

Il progetto di riferimento per questo sprint è it.unibo.ddrSystem.

1.1 Analisi dei requisiti

Questa sezione illustra tutti i passi necessari per svolgere la fase di analisi dei requisiti. Prima di tutto viene chiarito il significato di tutti i termini e concetti riportati nel file "Tema finale" (dato dal committente), cosicché l'analista abbia una chiara comprensione di ciò che il cliente si aspetta che il sistema faccia. Dopodiché è presentata una formalizzazione del sistema utilizzando i QActor: viene rappresentanto un modello per ogni componente del sistema. La fase di analisi dei requisiti comprende le seguenti domande:

1. **Di che sistema necessita il committente?**
   Il committente necessita di un sistema composto da un robot che deve esplorare tutta la hall di un aeroporto (R-explore) e scattare una foto (R-takePhoto) quando giunge in prossimità di una valigia. Ogni foto viene successivamente inviata ad un'altra parte del sistema denominata console (R-sendPhoto). Nel caso in cui la valigia risulti pericolosa un secondo robot deve provvedere al disinnescamento della bomba (R-reachBag). Si tratta dunque di un sistema distribuito eterogeneo.

2. **Da quanti componenti è composto il sistema?**
   Il sistema è formato da 3 componenti: 1 console e 2 robot (uno per l'esplorazione e l'altro per il disinnescamento). Anche se il committente ipotizza l'utilizzo di 2 robot fisici differenti, questi non saranno attivi contemporaneamente, dunque essi possono essere modellati come due comportamenti distinti dello stesso robot fisico.

3. **Qual è il compito della console?**
   Il compito della console è interpretare i comandi ricevuti dall'operatore e, nel caso in cui siano validi, inviarli al robot. Inoltre la console deve riuscire a comunicare con il robot, ad esempio, per ricevere informazioni sullo stato attuale del robot.

4. **Che cosa si intende per ostacolo?**
   Gli ostacoli sono entità sulle quali il robot non può passare perciò, ogni volta che il robot ne incontra uno, deve opportunamente evitarlo. Gli ostacoli modellati nel sistema sono:

   - valigie: lasciate dai passeggeri nella stanza al momento dell'evacuazione. Esse potrebbero essere disposte in 3 modi:
     - nel centro della stanza;
     - adiacenti a un muro;
     - in uno degli angoli della stanza.
   - muri della stanza.

5. **Quando il robot può partire con l'esplorazione?**
   Il robot può partire con l'esplorazione quando sono verificate due condizioni: il robot riceve un comando di inizio esplorazione e la temperatura della stanza è inferiore ad un certa soglia.

6. **Che cosa si intende per esplorazione autonoma di un robot?**
   Per esplorazione autonoma di un robot si intende la capacità di perlustrare interamente una stanza con ostacoli fissi (valigie e muri) e muovendosi lungo una superficie piana. Durante questa fase il robot deve far blinkare un led posto su di esso (R-blinkLed).

7. **Quando il robot si deve fermare?**
   Il robot si deve fermare in 3 casi:

   - quando riceve un comando di stop (R-stopExplore) dalla console;
   - quando si trova in prossimità di un ostacolo (R-stopAtBag);
   - quando la temperatura della hall supera la soglia fissata

8. **Quando il robot deve tornare alla base?**
   Il robot deve tornare alla base quando riceve dalla console il relativo comando (R-backHomeSinceBomb o R-backHome).

9. **Cosa fa il robot quando incontra un ostacolo?**
   Quando il robot incontra un ostacolo, si ferma (R-stopAtBag), fa la foto (R-takePhoto), la manda alla console (R-sendPhoto) e aspetta un comando dalla console prima di riprende l'esplorazione. Dopodiché il robot può: o tornare a casa e terminare quindi l'esplorazione (R-backHomeSinceBomb) o continuare l'esplorazione (R-continueExploreAfterPhoto) .

10. **Cosa fa il robot una volta terminata l'esplorazione della stanza?**
    Dopo che il robot ha controllato tutta la stanza senza trovare nessuna valigia sospetta, esso torna alla sua base.

11. **Quali informazioni deve conoscere la console riguardanti lo stato del robot?**
    La console deve avere delle informazioni riguardanti lo stato del robot per sapere come si sta muovendo (robot fermo, va avanti/indietro, ruota a destra/sinistra), in quale direzione e in che posizione si trova. Inoltre, deve poter ricevere le foto dei bagagli inviati dal robot e memorizzare le relative informazioni (data/orario e posizione del robot al momento dello scatto della foto).

12. **Cosa deve fare il robot se in fase di esplorazione la temperatura della hall supera una certa soglia?** Il robot si deve fermare nel punto in cui si trova e attendere che la temperatura della stanza diminuisca (R-TempOk) e che l'operatore ridia il comando di start (R-startExplore).

1.2 QActor formalisation

Una formalizzazione di quanto descritto nella sottosezione precedente la si può ottenere usando il linguaggio dei QActor. In particolare, si è realizzato un sistema composto da due attori (una console ed un robot) che operano nello stesso contesto.

```
System ddrSys

Context ctx ip [ host= "localhost" port=8078] -g green

QActor console context ctx {
  Plan init normal [
    println( "console initialised" )
  ]
}

QActor robot context ctx {
  Plan init normal [
    println( "robot initialised" )
  ]
}
```

1.3 Analisi del problema

Le problematiche emerse inizialmente dall'analisi dei requisiti e riguardanti i componenti sono:

1. **distribuzione**: il robot e la console sono fisicamente in due posti diversi, quindi il sistema deve essere distribuito;
2. **eterogeneità**: il robot e la console potrebbero utilizzare tecnologie diverse, quindi il sistema deve essere eterogeneo;
3. **interazione**: trattandosi di un sistema distribuito eterogeneo, il sistema deve essere message-based e event-based per permettere alle entità di interagire tra di loro;
4. **coordinazione**: il robot e la console devono coordinarsi tra loro, quindi è opportuno stabilire una policy per definire quando e come determinate azioni devono verificarsi.

Nella fig. 1 è rappresentato il primo modello del sistema. In questo modello è possibile osservare che vi sono 3 entità: operatore, console e robot.

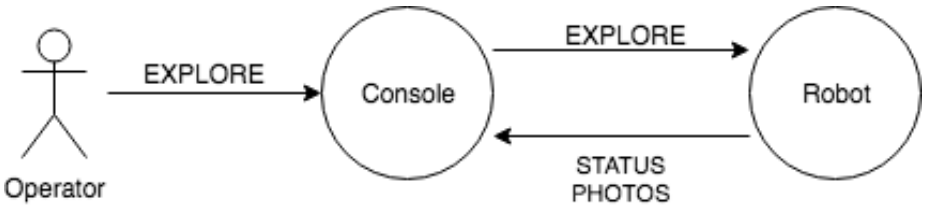

Figure 1: Il sistema osservato da un punto di vista esterno.

Analizzando più nel dettaglio il problema sono emerse le seguenti problematiche:

1. **Come riceve i comandi la console?**
   In questo sistema, l'interazione tra l'operatore e la console può essere modellata a procedure call: l'operatore preme un pulsante su un'interfaccia grafica, la console elabora il comando verificando che esso sia valido e infine chiama la procedura adatta alla sua gestione. Essendo tutto gestito internamente al componente console, risulta inutile utilizzare i paradigmi più complessi (ad esempio, message-based e event-based).
2. **Come interagiscono console e robot?**
   Essendo un sistema distribuito la console e il robot interagiscono attraverso lo scambio di messaggi. I messaggi scambiati tra le due entità sono formalizzati di seguito.

```
//Message from console to robot
Dispatch explore: explore(X)
Dispatch stopExplore: stopExplore(X)
Dispatch backHome: backHome(X)
Dispatch continueExplore: continueExplore(X)
Dispatch backHomeSinceBomb: backHomeSinceBomb(X)
Dispatch continueExploreAfterPhoto: continueExploreAfterPhoto(
    X)

//Message from robot to console
Dispatch sendPhoto: sendPhoto(X)

//Message from robot to robot
Dispatch reachBag: reachBag(X)


```

3. **Quali informazioni bisogna mantenere sul modello del robot?**
   Il modello del robot può essere descritto attraverso un insieme di informazioni riguardanti:

   (a) lo stato (per sapere se il robot è fermo, sta andando avanti o indietro, si sta ruotando a destra o sinistra);

   (b) la direzione (per sapere sapere come è orientato il robot all'interno della stanza: nord, sud, ovest, est);

   (c) la posizione (per sapere dove si trova il robot all'interno della stanza).

   In particolare, per rintracciare la posizione del robot all'interno della stanza, si utilizza il concetto di griglia, assumendo che esso si muova di passi unitari su di questa.

4. **Come fa la console ad avere le informazioni riguardanti lo stato del robot?**
   Ogni volta che il robot cambia stato emette un evento con le informazioni aggiornate riguardanti il nuovo stato assunto. Tali informazioni verranno poi inviate alla console. L'evento emesso è formalizzato di seguito.

```
Event modelContent: content(X)
```

   Lo stato del robot è inizialmente modellato utilizzando Prolog come:

```
//robot state formalisation
state(position(X,Y), direction(D), action(A))

position(X, Y)

direction(west)
direction(east)
direction(north)
direction(south)

action(moving)
action(stop)
```

```
13 action(take_picture)
14 action(send_photo)
```

5. **Come viene percepito il cambiamento di temperatura della stanza?** Di default si assume che la temperatura della stanza sia inferiore ad una certa soglia finché al robot non arriva il messaggio di "temperatureTooHigh", il quale indica che la soglia è stata superata. I messaggi inviati al robot sono i seguenti:

```
1 Dispatch temperatureTooHigh: temperatureTooHigh
2 Dispatch temperatureOk: temperatureOk
```

6. **Chi invia il messaggio della cambiamento della temperatura?** Ogni volta che la temperatura supera una certa soglia si scatena un evento che è percepito e gestito dalla console, la quale provvederà all'invio del messaggio "temperatureTooHigh" al robot.

7. **Come fa il robot ad esplorare la stanza?** La stanza viene esplorata in maniera autonoma dal robot, quest'ultimo costruisce progressivamente una mappa della stanza riportando su di essa ostacoli fissi, ossia le valigie e i muri, con le relative posizioni e dimensioni. Questo compito può essere suddiviso in due fasi:

   - esplorazione della stanza vuota.
   - esplorazione con ostacoli fissi.

8. **Quale strategia si può adottare per svolgere l'esplorazione della stanza vuota?** È possibile esplorare la stanza vuota in maniera incrementale: il robot coprirà dapprima una piccola area (a lui circostante) che mano a mano si espanderà fino a ricoprire l'intera stanza.

9. **Quale strategia si può adottare per svolgere l'esplorazione della stanza con ostacoli fissi?** È possibile esplorare la stanza con ostacoli fissi in maniera simile a quanto accadrebbe nel caso in cui la stanza fosse vuota: il robot, partendo dalla sua posizione iniziale, esplorerà dapprima la parte di stanza più vicina a lui per poi successivamente espandersi sempre di più. Ogni qual volta il robot si trovi in presenza di un ostacolo, si ricalcolerà un percorso per raggiungere la posizione desiderata sulla griglia. Inoltre, verranno memorizzate le informazioni relative all'ostacolo.

10. **Come fa il robot a riconoscere un ostacolo?** Assumiamo che il robot sia dotato nella parte frontale di un sonar e che ogni volta che il sonar rilevi un valore inferiore ad una certa soglia il robot si fermi in quanto si è in presenza di un ostacolo.

11. **Da quale prospettiva il robot scatta la foto all'ostacolo?** Il robot scatta la foto alla valigia esattamente dall'angolazione in cui esso si trova rispetto all'ostacolo nel momento in cui giunge in sua prossimità.

Infatti, l'angolazione della foto risulta ininfluente ai fini della valutazione della presenza o meno della bomba, poiché si suppone che il tool utilizzato dalla console esamini il bagaglio con una tecnologia a infrarossi.

12. **Come fa il robot a tornare nella posizione iniziale?**
    La prima cella , ossia (0,0), che il robot memorizza sulla mappa è la sua posizione iniziale, quindi basterà che esso percorra un qualunque tragitto dalla sua posizione attuale alla prima cella memorizzata della mappa per far sì che torni nella sua posizione iniziale.

13. **In caso di valigia sospetta, cosa fa il robot?**
    La valigia sospetta sarà l'ultimo ostacolo memorizzato nella mappa, in quanto quando questo viene rilevato la fase di esplorazione si sospende. Una volta individuata la valigia sospetta, il robot rientrà autonomamente alla base (R-backHomeSinceBomb) al robot. Quando quest'ultimo è tornato alla base (R-waitForHome), esso ripartirà per raggiungere la valigia sospetta, ossia l'ultima memorizzata sulla mappa (R-ReachBag), la inserirà poi in un contenitore e la trasporterà nella sua posizione iniziale (R-bagAtHome).

14. **Come fa la console ad interagire con il robot durante la fase di esplorazione?**
    Il robot dovrà avere una natura proattiva per gestire autonomamente l'esplorazione della hall e un natura reattiva per ricevere e rispondere prontamente ai messaggi della console.

In base all'analisi del problema, si è derivata l'architettura logica di figura 2. Al di sopra della linea sono rappresentate le entità principali che compongono il sistema e come queste interagiscono tra di loro , invece, al di sotto delle linea si trovano le relative implementazioni di tali entità.

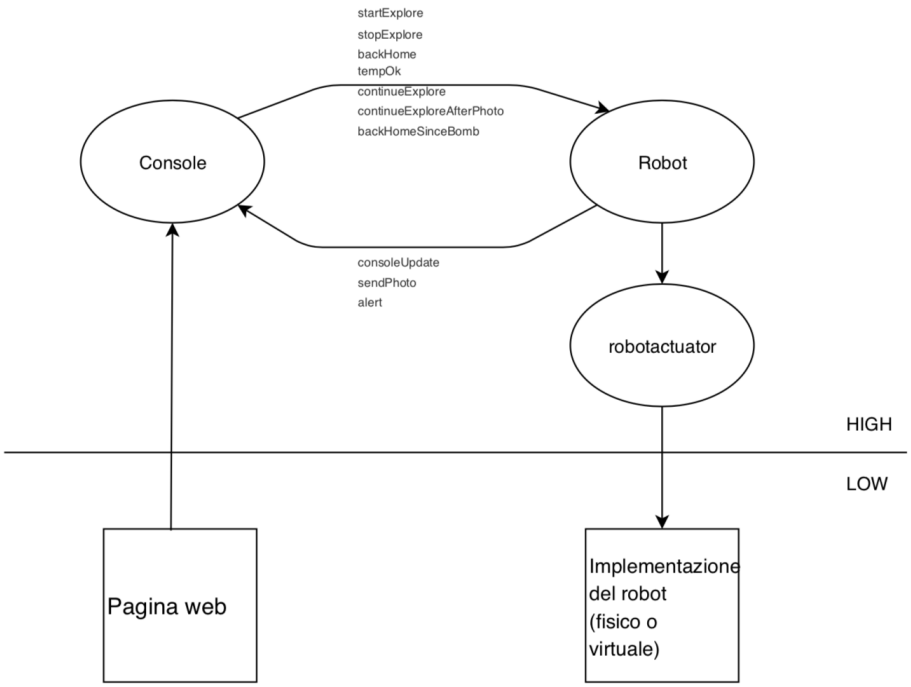

Figure 2: L'architettura logica del sistema.

1.4 Test Plan

Nel Test Plan bisogna verificare che:

- quando l'utente spinge il pulsante di startExploration e la temperatura della stanza è inferiore ad una soglia fissata, il robot inizi l'esplorazione.
- quando la console percepisce l'evento del cambio di temperatura, verifica se è ancora adeguata e se è troppo alta invia al robot il messaggio di stopExplore
- quando il robot riceve il messaggio di explore inizia ad esplorare tutta la stanza;
- quando il robot riceve il messaggio di stopExplore si ferma nel punto in cui si trova;
- quando il robot riceve il messaggio di backHome torna nella sua posizione iniziale;
- quando il robot incontra una valigia non ancora analizzata si ferma, gli scatta una foto e la spedisce con un messaggio alla console (sendPhoto).
- quando il robot riceve il messaggio di continueExplore riprende l'esplorazione.

- quando il robot riceve il messaggio di comeBackSinceBomb torna alla base e passa alla fase di recupero della valigia pericolosa.

```
System ddrSys

//robot state update event
Event consoleUpdate: consoleUpdate(X)

//temperature change event
Event tempOk: tempOk(X)

//Message from console to robot
Dispatch explore: explore(X)
Dispatch stopExplore: stopExplore(X)
Dispatch backHome: backHome(X)
Dispatch continueExplore: continueExplore(X)
Dispatch backHomeSinceBomb: backHomeSinceBomb(X)
Dispatch continueExploreAfterPhoto: continueExploreAfterPhoto(X)

//Message from robot to console
Dispatch sendPhoto: sendPhoto(X)

//Message from robot to robot
Dispatch reachBag: reachBag(X)


Context ctx ip [host="localhost" port=8078] -g cyan

QActor console context ctx {

  Plan init normal [
    println("Console intialized")
  ]
}

QActor robot context ctx {
  Plan init normal [

    println("Robot intialized")
  ]
}
```

1.4.1 Work Plan

Per la realizzazione dell'intero sistema si procederà in maniera incrementale, raggiungendo ad ogni sprint un determinato obiettivo. Gli obiettivi individuati sono i seguenti:

1. creazione di un sistema con un robot in grado di eseguire dei comandi di spostamento (movingForward, movingBackward, rotateLeft, rotateRight, stopped).

2. creazione di un sistema in cui il robot riesca a scambiare messaggi con una web page;

3. creazione di un sistema con un robot in grado di raggiungere un punto specifico riportato sulla mappa.

4. creazione di un sistema con un robot in grado di esplorare una stanza vuota per poi crearne una mappa. L'esplorazione prevede l'utilizzo di una strategia;

5. creazione di un sistema con un robot in grado di esplorare una stanza contenente ostacoli fissi per poi crearne una mappa;

Tale sprint ed i seuguenti sono suddivisi nelle fasi di analisi dei requisiti e analisi del problema, creazione del test plan, modellazione ed implementazione del sistema.

2 Sprint 1

Il progetto di riferimento per questo sprint è it.unibo.ddrSystem1.

OBIETTIVO: creazione di un sistema con un robot in grado di rispondere a dei comandi. Il robot partendo da una posizione iniziale, dovrà muoversi avanti e indietro, ruotare a destra e sinistra e fermarsi. Inoltre, il robot dovrà rilevare la presenza delle pareti (ostacoli fissi).

2.1 Work Plan

1. progettazione del test plan per verificare che il robot si muova correttamente in presenza o meno di ostacoli (muro).
2. definizione dei messaggi e degli eventi gestiti dal sistema: messaggi per dare i comandi di movimento e l'evento del sonar del robot;
3. definizione dell'interazione tra il QActor console e il QActor robot;
4. definizione del comportamento del robot quando percepisce un evento SonarRobot.

2.2 Analisi dei requisiti

Per la creazione di questo sistema si utilizza un robot dotato di un sonar per rilevare eventuali ostacoli di fronte ad esso. Il robot si muove su una griglia e gli spostamenti del robot sono modellati con un passi unitari, in quanto, ad ogni passo, esso si muove esattamente di una cella. Le dimensioni della cella equivalgono a quelle del robot.

2.3 Test plan

I test da fare sul sistema sono:

`` @Test fun initialStateTest(){ println("%%% initialStateTest %%%") solveCheckGoal(robot!!, "model(actuator, robot, state(stopped), direction(south), position(0,0))") printRobotState() } ```	Verificare che il robot, prima dell'inizio dell'esplorazione, sia fermo, orientato verso sud e che si trovi nella posizione iniziale (0,0).
`` @Test fun moveTest() { println("%%% moveTest %%%") rotateRight() delay(700) rotateLeft() delay(500) moveForward() delay(700) moveBackward() delay(700) stoprobot() solveCheckGoal(robot!!, "model(actuator, robot, state(stopped), direction(south), position(0,0))") printRobotState() } ```	Test per verificare che i comandi per spostare il robot vengano eseguiti in maniera corretta.
`` @Test fun wallDetectingTest() { moveForward() //no obstacle assumed moveForwardWithWall() solveCheckGoal(robot!!, "model(actuator, robot, state(stopped), _, _)") printRobotState() } ```	Verificare che il robot, una volta riconosciuta la presenza di una parete, vada indietro e poi si fermi.

2.4 Model

Per la realizzazione di questo sprint si è preso come punto di partenza il sistema modellato nello sprint 0, in cui è stato definito più nel dettaglio il comportamento del robot. In particolare si è modellato un robot in grado di ricevere ed eseguire determinati comandi. In questo prototipo le azioni vengono eseguite su una base di conoscenza prolog in modo da tener sempre aggiornato il modello. Il modello è rappresentato come un fatto prolog il quale viene aggiornato ad ogni azione invocata dal sistema. Le azioni sono modellate come predicati prolog.
Il modello è: model(actuator, robot, state(S), direction(D), position(X,Y)). In cui

- S, è la variabile che rappresenta lo stato del sistema e può assumere i seguenti valori: stopped, movingForward, movingBackward, rotateLeft, rotateRight.
- D, è la direzione del sistema e può assumere i seguenti valori: north, east, south, ovest.
- X,Y sono le coordinate che rappresentano la cella in cui si trova il sistema e possono essere qualsiasi combinazioni di interi compresi tra 0 e il numero massimo di celle della stanza.

Le azioni sono :action(robot, move(M)) :- changeModel(actuator, robot, movingForward). In cui M è una variabile che può assumere valori diversi a seconda dell'azione che si vuol far eseguire al sistema, può assumere i seguenti valori:

- w: se si vuole che il robot si muova in avanti
- s: se si vuole che il robot si muova indietro
- d: se si vuole che il robot si giri a destra
- a: se si vuole che il robot si muova a sinistra
- h: se si vuole che il robot si fermi

```
Event  sonarRobot: sonarRobot(DISTANCE)      //from
    sonar on robot
Dispatch: userCmd( CMD ) //Message from console to
    robot
Dispatch robotCmd: robotCmd (CMD) //Selfsending robot
    message
```

A fianco sono riportati i messaggi e l'evento che sono stati utilizzati all'interno del sistema.

- sonarRobot: evento che si scatena quando il robot si trova in prossimità di una parete
- userCmd: messaggio che viene inviato dalla console al robot per far sì che esso cambi il suo stato
- robotCmd: messaggio che il robot manda a se stesso nel caso debba effettuare degli spostamenti per evitare un ostacolo.

```
QActor robot context ctx {
["var obstacle = false"]
  State s0 initial {
    solve (consult ("ddrsys.pl"))
    solve (consult ("resourceModel.pl"))
    println("Robot intialized")

  }
  Goto waitForEvents

  State waitForEvents {   }

  Transition t0   whenMsg userCmd  -> handleCmd
          whenMsg robotCmd -> handleCmd
          whenEvent sonarRobot -> handleSonarRobot

  State handleCmd{
    printCurrentMessage
    onMsg (userCmd    : userCmd( CMD )){
      solve( action( robot, move($payloadArg(0)) ) ) //
    change the robot state model
    }
    onMsg (robotCmd    : robotCmd( CMD )){
      solve( action( robot, move($payloadArg(0)) ) ) //
    change the robot state model
    }
  }

  Goto waitForEvents

  State handleSonarRobot{
    printCurrentMessage
    onMsg ( sonarRobot : sonarRobot(DISTANCE) ){
      ["obstacle = Integer.parseInt( payloadArg(0) ) <
    10 "]
    }
  }

  Goto handeObstacle  if "obstacle" else waitForEvents

  State handeObstacle{
    println("handleObstacle: going backward")
    forward robot -m robotCmd : robotCmd( s )
      //UPDATE the model : supported action
      //run itunibo.robot.resourceModelSupport.
    updateModel( myself, "s" )
    delay 300
    println("handeObstacle: stopping")
      forward robot -m robotCmd : robotCmd( h )
      //UPDATE the model : supported action
      //run itunibo.robot.resourceModelSupport.
    updateModel( myself, "h" )
    }

  Goto waitForEvents

}
```

Il robot è formato da tre stati principali:

- waitForEvents: nel quale rimane in attesa: di un comando inviato dall'utente (userCmd), di un comando inviato da se stesso (robotCmd), di un evento scatenato dal proprio sonar (sonarRobot) quando si trova in prossimità di un ostacolo.
- handleCmd: nel quale il robot gestisce le due tipologie di comandi, in questo caso esegue la stessa azione, ovvero cambia solamente la base di conoscenza del sistema
- handeObstacle: nel quale è definita la logica di comportamento a seguito della rilevazione di un ostacolo: si fa andare un po' indietro il robot e poi lo ferma.

3 Sprint 2

Il progetto di riferimento per questo sprint è it.unibo.ddrSystem2.

OBIETTIVO: creazione di un robot in grado di esplorare una stanza rettangolare vuota. Il robot, durante l'esplorazione della stanza, dovrà essere in grado di costruire incrementalmente una mappa della stanza.

3.1 Work Plan

1. utilizzare il simulatore di Soffritti (robot virtuale) per verificare che il sistema creato funzioni correttamente. Nell'effettuare il collegamento tra i due si rimarrà technology independent (robotSupport);

2. utilizzare un robot fisico (realnano) per verificare che il sistema creato funzioni correttamente e che la scelta tecnologica non impatti sull'architettura logica del sistema. Il robot deve essere in grado di muoversi in avanti, in indietro, ruotare a destra, a sinistra e fermarsi, proprio come quello simulato;

3. definizione della strategia di esplorazione della stanza.

4. progettazione del test plan per verificare che il robot riesca a costruire correttamente la mappa della stanza seguendo la strategia scelta.

5. progettazione di una strategia per la creazione della mappa relativa alla stanza. Valutare le diverse possibilità: utilizzo di una base di conoscenza prolog, creazione di una libreria per la gestione della mappa, utilizzo di plannerUtils già fornite.

3.2 Analisi dei requisiti

Cosa si intende per esplorazione della stanza?
Per esplorazione si intende muovere il robot in maniera organizzata e autonoma dentro ad una stanza finché non sono state esplorate tutte le celle. Ad esempio, un risultato che si potrebbe ottenere a fine esplorazione è il seguente:
r, 1, X,
1, 1, X,
X, X, X

Quali informazioni raccoglierà in fase di esplorazione?
Il robot raccoglie le informazioni riguardanti la stanza, in particolare, quali celle sono state visitate, quali ancora no e dove si trovano i muri.

Quando il robot inizia ad esplorare autonomamente la stanza?
Quando riceve il comando di "start" (R-startExplore) dalla console.

Quando il robot smette di esplorare autonomamente la stanza?
Quando riceve il comando di "stop" (R-stopExplore) dalla console.

3.3 Analisi del problema

Esistono varie strategie per permettere al robot di raggiungere l'obiettivo, ognuna di queste prevede che la posizione iniziale (base) del robot coincida con uno degli angoli della stanza e che la distanza percorsa dal robot ad ogni spostamento sia unitaria (in questo caso, l'unità di riferimento è la dimensione del robot). Le strategie vagliate sono le seguenti:

- **a chiocciola**: utilizzando questa strategia il robot esplora in maniera incrementale tutta la superficie della stanza, inizialmente il robot percorre la parte di stanza a lui strettamente adiacente per poi, a mano a mano, espandersi sino ad esplorarla per intero.
- **a colonne/righe**: utilizzando questa strategia il robot esplora la stanza muovendosi inizialmente lungo un lato finché non incontra il muro opposto. Una volta incontrato si gira di 90° nella direzione in cui non sono presenti muri. Poi si sposta di una unità, si rigira di 90° e procede dritto fino a che non rincontra un altro muro. Al termine dell'esecuzione il robot si deve trovare o nell'angolo opposto (nel caso in cui il numero di colonne/righe fosse dispari) oppure nell'angolo adiacente (nel caso in cui il numero di colonne/righe fosse pari). Il modo per implementare questa strategia potrebbe essere quello di creare un robot che percorra tutta la lunghezza di un solo lato della stanza.
- **a spirale**: utilizzando questa strategia il robot si muove dapprima lungo le quattro pareti (in maniera oraria e sempre lungo la parente adiacente successiva rispetto a quella che è appena stata esaminata), dopodiché effettua il medesimo tragitto ma restringendo il campo da esplorare. L'esplorazione procede per rettangoli concentrici via via sempre più piccoli e termina quando il robot si trova nel centro della stanza.

Esistendo diverse strategie per realizzare questo compito si può pensare di incapsulare la logica di comportamento in più componenti esterni per poi utilizzare quello desiderato.

3.4 Model

Si è scelta la strategia della chiocciola per l'esplorazione autonoma in quanto si ritiene che essa sia quella che ottimizza il ritrovamento della bomba, poiché permette di effettuare una ricerca più omogenea e quindi di ritrovare la bomba as soon as possible. Dato che si ritiene importante dividere la logica di esplorazione del robot dall'attuazione di essa, partendo dal QActor `robot` modellato nello sprint precedente, si è deciso di suddividerlo in due QActor differenti, ossia:

- `robotmind`: ha il compito di pianificare le azioni necessarie per raggiungere una determinata posizione sulla mappa (vedi funzione `setGoal(X,Y)`); posizione che diventerà incrementalmente sempre più lontana dalla quella iniziale in quanto si è scelto di esplorare la stanza con la strategia della

chiocciola e la cui massima distanza dalla base coinciderà con l'angolo opposto della stanza. Una volta pianificate le azioni per raggiungere un punto della stanza, queste verranno eseguite una alla volta e, in presenza di un ostacolo (muro), una delle azioni fallirà e comporterà il ricalcolo del tragitto che il robot dovrà percorrere (`vedi State setGoalAfterWall`).

Il codice di robotmind è riportato nel Listing 1.

- `robotactuator`: ha il compito di eseguire una alla volta le azioni (vedi `move(msg(M))` pianificate da `robotmind`.

  Il codice di robotactuator è riportato nel Listing 2.

Analizzando i movimenti del robot si è evidenziato che il robot può incontrare un muro solamente quando si muove in avanti, dunque si è introdotto un terzo QActor: `onestepahead`. In particolare, questo attore riceverà un messaggio `onestep` (vedi funzione `attemptToMoveAhead()`) che farà muovere il robot in avanti e controllerà se effettivamente il movimento è possibile. Se il movimento è possibile, in quanto non vi sono ostacoli, ciò verrà notificato a `robotmind` con un messaggio di `stepOk` altrimenti `robotmind` riceverà un messaggio di `stepFail`. Il codice di onestepahead è riportato nel Listing 3.

L'architettura del sistema è riportata in fig. 3

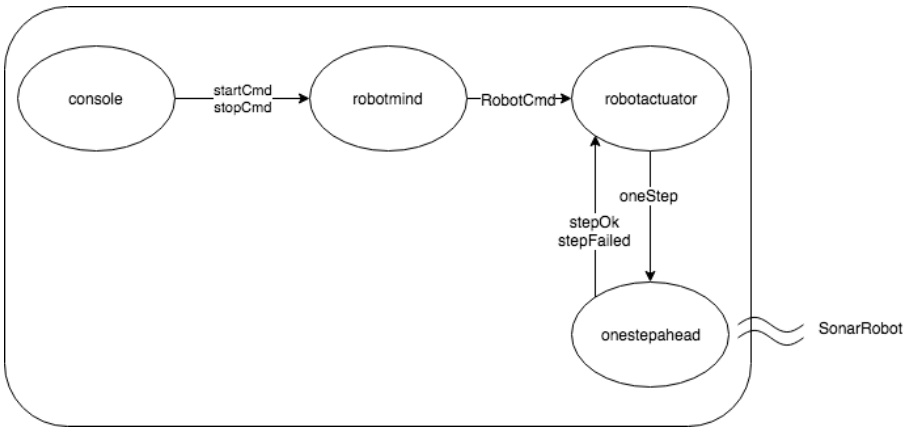

Figure 3: Architettura logica del sistema dello sprint 2.

```
1
2  QActor robotmind context ctx {
3  ["var Curmove      = \"\"
4  var IterCounter = 0
5  var backHome = false
6  var maxX = 0
7  var maxY = 0
8  var finish = false
9
10 //VIRTUAL ROBOT
```

```
11 var StepTime   = 330
12
13 var Tback       = 0
14 "]
15   State s0 initial {
16     println("&&&  robotmind STARTED")
17     solve (consult ("ddrsys.pl"))
18     solve (consult ("resourceModel.pl"))
19     println("Robot intialized")
20     run itunibo.planner.plannerUtil.initAI()
21     println("INITIAL MAP")
22     run itunibo.planner.plannerUtil.showMap()
23
24   }
25
26   Goto waitForStart
27
28   State waitForStart {  printCurrentMessage }
29
30   Transition t0  whenMsg startCmd  -> startExploration
31            whenMsg startTest -> startExplorationTest
32
33   State startExplorationTest {
34     ["finish = true
35       backHome = false
36
37       var x = \"\"
38       var y = \"\"  "]
39     println("&&&  exploration TEST")
40
41     printCurrentMessage
42     onMsg( startTest : startTest(X, Y) ) {
43       [" x =payloadArg(0)
44          y =payloadArg(1) "]
45     run itunibo.planner.plannerUtil.setGoal(x,y)
46     run itunibo.planner.moveUtils.doPlan( myself ) //moves stored
       in actor kb
47     }
48   }
49   Goto doPlan
50
51
52   State startExploration {
53     println("&&&  exploration STARTED")
54     run itunibo.planner.plannerUtil.setGoal("1","1")
55     run itunibo.planner.moveUtils.doPlan( myself ) //moves stored
       in actor kb
56   }
57
58   Goto doPlan
59
60   //raggiungo la cella
61   State doPlan {
62     run itunibo.planner.plannerUtil.showMap()
63     solve( retract( move(M) ) )    //consume a move
64     ifSolved {  ["Curmove = getCurSol(\"M\").toString()"]  }
65      else { ["Curmove=\"nomove\" "]  }
```

```
66    }
67
68    Goto handlemove if "(Curmove != \"nomove\")" else choose
69
70    State handlemove {}
71
72    Goto domove if "(Curmove != \"w\")" else attempttogoahead
73
74    State domove {
75      run itunibo.planner.moveUtils.doPlannedMove(myself, Curmove)
76      forward robotactuator -m robotCmd : robotCmd(\$Curmove)
77      delay 700
78      forward robotactuator -m robotCmd : robotCmd(h)
79    }
80
81    Goto doPlan
82
83    //roomboundaryplanning.qak
84    State attempttogoahead {
85      run itunibo.planner.moveUtils.attemptTomoveAhead(myself,
      StepTime)
86    }
87    Transition t0   whenMsg stepOk   -> stepDone
88            whenMsg stepFail -> stepFailed
89
90    State stepDone{
91      run itunibo.planner.moveUtils.doPlannedMove(myself, "w")
92    }
93
94    Goto doPlan
95
96    State stepFailed{
97      println("&&&  FOUND WALL")
98  ["var TbackLong = 0L"]
99
100     //printCurrentMessage
101     onMsg( stepFail : stepFail(Obs, Time) ) {
102       ["Tback=payloadArg(1).toLong().toString().toInt() / 2
103       TbackLong = Tback.toLong()"]
104       println("stepFailed \${payloadArg(1).toString()}")
105     }
106
107     println(" backToCompensate stepTime=\$Tback")
108     forward robotactuator -m robotCmd : robotCmd(s)
109     delayVar TbackLong
110     forward robotactuator -m robotCmd : robotCmd(h)
111     delay 700
112   //-------------------------------------------------
113     run itunibo.planner.plannerUtil.wallFound()
114
115   }
116
117   //Goto endOfJob //**checkWallTest
118   Goto setGoalAfterWall
119
120   State setGoalAfterWall{
121     solve( retractall( move(_) ))
```

```
122   ["
123   if( itunibo.planner.plannerUtil.getDirection() == \"downDir\" ){
124     maxY = itunibo.planner.plannerUtil.getPosY()
125     if(maxX == 0 ){
126       itunibo.planner.plannerUtil.setGoal(IterCounter, maxY)
127     } else {itunibo.planner.plannerUtil.setGoal(maxX, maxY)}
128   }
129   else if( itunibo.planner.plannerUtil.getDirection() == \"rightDir
        \" ){
130     maxX = itunibo.planner.plannerUtil.getPosX()
131     if (maxY == 0 ){
132       itunibo.planner.plannerUtil.setGoal(maxX, IterCounter)
133     } else { itunibo.planner.plannerUtil.setGoal(maxX, maxY) }
134   } else {
135     itunibo.planner.plannerUtil.setGoal(0, 0)
136   }
137   "]
138     run itunibo.planner.moveUtils.doPlan( myself )
139   }
140
141   Goto doPlan
142
143
144   State choose {}
145   Goto goBackHome if "backHome" else nextStep
146
147   //torno a casa
148   State goBackHome{
149   ["backHome = false"]
150     println("&&&  returnToHome")
151     //solve( retractall( move(_) ))   //clean the actor kb
152     run itunibo.planner.plannerUtil.setGoal(0,0)
153       run itunibo.planner.moveUtils.doPlan( myself )
154
155       delay 700
156     }
157
158   Goto doPlan
159
160   State nextStep {}
161
162   Goto endOfJob if "finish" else calculatenextstep
163
164   State calculatenextstep{
165 ["IterCounter++
166   backHome = true
167   if (maxX == 0 && maxY == 0){ itunibo.planner.plannerUtil.setGoal(
        IterCounter,IterCounter) }
168   else if( maxX != 0 && maxY == 0 ){ itunibo.planner.plannerUtil.
        setGoal(maxX,IterCounter) }
169   else if( maxX == 0 && maxY != 0 ){ itunibo.planner.plannerUtil.
        setGoal(IterCounter, maxY) }
170   else {
171     itunibo.planner.plannerUtil.setGoal(maxX, maxY)
172     finish = true
173 }
174 "]
```

```
    println("&&&  nextStep")
    run itunibo.planner.moveUtils.doPlan( myself )
  }
  Goto doPlan

  State endOfJob{
    ["if (maxX != 0 && maxY != 0) {itunibo.planner.plannerUtil.
    fixwalls(maxX, maxY)}"]

    println("FINAL MAP")
    run itunibo.planner.plannerUtil.showMap()
    println("&&&  planex0 ENDS")
  }

}
```

Listing 1: Codice di QActor robotmind in ddrSystem2

```

QActor robotactuator context ctx {
  State s0 initial {
    ["
//CREATE A PIPE for the sonar-data stream
val filter = itunibo.robot.sonaractorfilter( \"sonaractorfilter\" ,
     myself  )
val logger = itunibo.robot.Logger(\"logFiltered\")
filter.subscribe(logger)

"]

      solve( consult("basicRobotConfig.pl") )
    solve( robot(R, PORT) )  //R = virtual | realmbot | realnano
      ifSolved {
        println( "USING ROBOT : \${getCurSol(\"R\")},  port= \${
    getCurSol(\"PORT\")} " )
        run itunibo.robot.robotSupport.create( myself, @R, @PORT,
    filter )
      }
      else{ println("no robot") }

      run itunibo.robot.robotSupport.move( "msg(a)" )
      delay 700
      run itunibo.robot.robotSupport.move( "msg(d)" )
      delay 700
      run itunibo.robot.robotSupport.move( "msg(h)" )
  }
  Goto waitCmd

  State waitCmd{  } //robotCmd comes from a console OUTSIDE this (
    sub)system
  Transition t0  whenMsg   robotCmd  -> handleRobotCmd

  State handleRobotCmd{ //does not handle alarms
    printCurrentMessage
    onMsg( robotCmd : robotCmd( MOVE ) ) { //MOVE = w | a | s | d |
     h
```

```
35        run itunibo.robot.robotSupport.move( "msg(\${payloadArg(0)})"
         )
36      }
37    }
38    Goto waitCmd
39 }
```

Listing 2: Codice di QActor robotactuator in ddrSystem2

```
1
2  QActor onestepahead context  ctx {
3  ["
4  var foundObstacle = false;
5  var StepTime = 0L;
6  var Duration=0
7  "]
8    State s0 initial {
9      ["foundObstacle = false "]
10   }
11   Transition t0 whenMsg onestep -> doMoveForward
12
13   State doMoveForward{
14     onMsg( onestep : onestep( TIME ) ) {
15       ["StepTime = payloadArg(0).toLong()"]
16       forward robotactuator -m robotCmd : robotCmd(w)
17       ["startTimer()"] //startTimer is built-in in the actor
18
19     }
20   }
21   Transition t0 whenTimeVar StepTime  -> endDoMoveForward
22                 whenEvent sonarRobot -> stepFail
23
24   State endDoMoveForward{
25     forward robotactuator -m robotCmd : robotCmd(h)
26     forward robotmind -m stepOk : stepOk
27   }
28   Goto s0
29
30
31
32   State stepFail{
33     ["Duration=getDuration()"]  //getDuration is built-in in the
       actor
34     printCurrentMessage
35     println("onestepahead stepFail Duration=$Duration ")
36
37     forward robotmind -m stepFail : stepFail(obstacle, $Duration)
38   }
39   Goto s0
40 }
```

Listing 3: Codice di QActor onestepahead in ddrSystem2

3.5 Test plan

I test da fare sul sistema sono:

`` @Test fun cheGoalTest() { GlobalScope.launch{ console!!.forward("startTest", "startTest(1,1)", "robotmind") } delay(10000) val pos = getRobotPos() assertTrue(pos == "(1,1)") printRobotState() } ```	verificare che il robot virtuale riesca a raggiungere un determinato obiettivo (i.e. cell(1,1)) .
`` @Test fun checkWallTest() { GlobalScope.launch{ console!!.forward("startTest", "startTest(0,8)", "robotmind") } delay(5000) val state = getRobotState() assertTrue(state == "downDir, (0,7)") printRobotState() } ```	verificare che il robot virtuale, una volta riconosciuta la presenza di una parete, si riposizioni nella cella immediatamente precedente e poi si fermi.
`` @Test fun finalMapTest() { var testRoomMap = """|r, 1, 1, 1, 1, 1, 1, X, |1, 1, 1, 1, 1, 1, 1, X, |1, 1, 1, 1, 1, 1, 1, X, |1, 1, 1, 1, 1, 1, 1, X, |1, 1, 1, 1, 1, 1, 1, X, |1, 1, 1, 1, 1, 1, 1, X, |1, 1, 1, 1, 1, 1, 1, X, |1, 1, 1, 1, 1, 1, 1, X, |X, X, X, X, X, X, X, X, """ GlobalScope.launch{ console!!.forward("startCmd", "startCmd", " robotmind") } delay(99000) assert(itunibo.planner.plannerUtil.getMap() == testRoomMap) } ```	verificare la correttezza della mappa prodotta dal robot al termine dell'esplorazione: quest'ultima dovrà combaciare con quella effettiva della stanza per quanto riguarda dimensioni e numero di celle.

4 Sprint 3

I progetti di riferimento per questo sprint sono:

- it.unibo.ddrSystem3 (lo stato del robot è modellato con prolog e visualizzato sulla web page)
- it.unibo.ddrSystem4 (lo stato del robot è modellato con prolog e CoAP e visualizzato sulla web page)
- it.unibo.frontend (interfaccia web)

OBIETTIVO: rendere le informazione del sistema (robot e stanza) accessibili via web utilizzando il protocollo standard RESTful.

4.1 Work Plan

1. rendere le informazioni del sistema accessibili via web;
2. creazione di un'interfaccia node dove:
   - inviare comandi al sistema (start, ossia (R-startExplore) e stop, ossia R-stopExplore);
   - visualizzare lo stato del sistema (in particolare lo stato del robot e le informazioni da lui raccolte, ossia R-consoleUpdate);

4.2 Analisi del problema

Le informazioni del sistema che devono essere visualizzate sulla web page riguardano il robot e la stanza. Fino a questo momento lo stato del robot è stato gestito mediante una base di conoscenza prolog (`resourceModel.pl`). Questa prima scelta è stata effettuata in quanto la risorsa era acceduta solamente dall'interno del sistema (test). Per quanto riguarda la stanza, essa è stata finora modellata in Java mediante la libreria `it.unibo.planner`. Nasce ora l'esigenza di accedere a tali risorse mediante internet rendendo quindi visibile lo stato del robot e quello di avanzamento di esplorazione della stanza sulla pagina web (`frontend.js`). Di conseguenza viene spontaneo modellare queste risorse utilizzando il protocollo standard RESTful. La tecnologia scelta a tale scopo è CoAP.

4.3 Model

Volendo dare la possibilità di accedere alle informazioni del sistema via web il sistema può essere considerato un sistema IoT. Dunque si ritiene importante utilizzare un'architettura di tipo esagonale in cui il modello delle risorse è al centro e ogni cambiamento del sistema avviene conseguentemente ad una modifica del modello. Si è introdotto dunque un nuovo QActor (resourcemodel) responsabile della gestione del modello delle risorse. L'interazione tra la web page e il sistema verrà gestita anche utilizzando un approccio di tipo publish/subscribe, in particolare utilizzando MQTT.

Table 1: QActor resourcemodel in exploration.qak

Codice	Descrizione
1 2 `Actor resourcemodel context robotResourceCtx{` 3 4 `State s0 initial {` 5 `solve( consult("sysRules.pl") )` 6 `solve( consult("resourceModel.pl") )` 7 `solve( showResourceModel )` 8 `run itunibo.coap.modelResourceCoap. create( myself, "resourcemodel" )` `//CoAP access` 9 `}` 10 `Goto waitModelChange` 11 12 `State waitModelChange{ }` 13 `Transition t0 whenMsg modelUpdate -> updateModel //forward from robotmind` 14 15 `State updateModel{` 16 `printCurrentMessage` 17 `onMsg( modelUpdate : modelUpdate( robot,V ) ) {` 18 19 `run itunibo.robot. resourceModelSupport. updateRobotModel( myself, payloadArg(1) )` 20 `solve( showResourceModel )` 21 `}` 22 `onMsg( modelUpdate : modelUpdate( sonarRobot,V ) ) {` 23 `run itunibo.robot. resourceModelSupport. updateSonarRobotModel( myself, payloadArg(1) )` 24 `}` 25 `onMsg( modelUpdate : modelUpdate( roomMap,V ) ) { //JULY19` 26 `//println("modelUpdate roomMap")` 27 `run itunibo.robot. resourceModelSupport. updateRoomMapModel( myself, payloadArg(1) )` 28 `}` 29 `}` 30 `Goto waitModelChange` 31 `}`	il QActor resourcemodel inzialmente crea la risorsa CoAP. Il CoAP server, una volta fatto partire, renderà accessibile tale risorsa all'indirizzo `coap://localhost:5683/resourcemodel` ed i CoAP client (i.e. la web page) potranno essere notificati quando lo stato della risorsa cambia (`fun updateState( modelitem : String )` nella classe `it.unibo.coap.modelResourceCoap.kt`) Ogni qualvolta il QActor resourcemodel riceve da QActor robotmind un messaggio di `modelUpdate` Il QActor resourcemodel prima emetterà un evento `modelContent` per notificare la web page (gestito tramite MQTT) ed aggiornarla con le nuove informazioni, poi procederà con l'aggiornamento della risorsa CoAP. Il messaggio di modelUpdate può corrispondere ad uno dei seguenti formati: • `modelUpdate : modelUpdate(robot,V )` • `modelUpdate : modelUpdate(sonarRobot,V )` • `modelUpdate : modelUpdate(roomMap,V )`

Continued on next page

Table 1 – *Continued from previous page*

Codice	Descrizione
1 2 3 `QActor robotmind context robotMindCtx {` 4 5 `...` 6 7 `State waitForStart {` 8 `}` 9 10 `Transition t0 whenMsg startCmd -> startExploration` 11 12 13 `State startExploration {` 14 `println("&&& exploration STARTED")` 15 `run itunibo.planner.plannerUtil.setGoal("1","1")` 16 `run itunibo.planner.moveUtils.doPlan( myself ) //moves stored in actor kb` 17 `}` 18 19 `...`	Per inviare il comando di "start" si è reso necessario far comunicare la web page con il QActor robotmind. In particolare, quando verrà premuto il bottone "start" sulla web page, il publisher (ossia le web page) pubblicherà il messaggio `msg(startCmd,dispatch,js,robotmind, startCmd,1)` sulla topic `unibo/qak/robotmind` (topic alla quale il QActor robotmind ha fatto la subscribe al momento della sua creazione)) dell'MQTT broker. Dopodiché, il QActor robotmind, ricevuto il messaggio `startCmd`, darà il via all'esplorazione automatizzata della stanza così come la si era strutturata nello sprint precedente (punto 3 del work plan).

Continued on next page

Table 1 – *Continued from previous page*

Codice	Descrizione
`QActor robotmind context robotMindCtx {` `...` `//raggiungo la cella` `State checkStop { }` `Transition t1 whenTime 100 -> doPlan` `whenMsg stopCmd ->` `handleStop` `...` `State handleStop{` `onMsg(stopCmd : stopCmd) {` `forward robotactuator -m robotCmd : robotCmd(h)` `forward resourcemodel -m modelUpdate : modelUpdate(robot, h)` `}` `}` `Transition t3 whenMsg startCmd -> doPlan` `}`	Similmente accade per il comando di "stop", in quanto il publisher (ossia le web page) pubblicherà il messaggio `msg(stopCmd,dispatch,js,robotmind, stopCmd,1)` sulla topic `unibo/qak/robotmind` dell'MQTT broker. Dopodiché, il QActor robotmind, ricevuto il messaggio `stopCmd`, arresterà l'esplorazione. Sarà possibile riprendere l'esplorazione premendo sul comando "start".
`QActor resourcemodel context robotResourceCtx{` `...` `State updateModel{` `...` `onMsg( modelUpdate : modelUpdate( robot,V ) ) {` `run itunibo.robot.resourceModelSupport.updateRobotModel( myself, payloadArg(1) )` `solve( showResourceModel )` `}` `...` `}`	Per visualizzare le informazioni relative al robot e alla stanza, si utilizza la topic`unibo/qak/events` sulla quale il QActor resourcemodel (publish) pubblicherà l'evento `( "modelContent" , "content(robot($RobotState, $RobotDir, $RobotPos ))")` ogni qualvolta riceva un messaggio `forward resourcemodel -m modelUpdate : modelUpdate( robot, $Curmove)` dal QActor robotmind. Dopodichè, la web page (subscribe), verrà notificata del cambiamento delle informazioni e provvederà ad aggiornarle.

Continued on next page

Table 1 – *Continued from previous page*

Codice	Descrizione
1 2 `QActor resourcemodel context robotResourceCtx{` 3 `...` 4 `State updateModel{` 5 `...` 6 `onMsg( modelUpdate : modelUpdate( roomMap,V ) )` 7 `run itunibo.robot. resourceModelSupport. updateRoomMapModel( myself, payloadArg(1) )` 8 `}` 9 `...` 10 `}`	Il meccanismo attraverso il quale vengono visualizzate le info raccolte dal robot in merito all'esplorazione della stanza rimane invariato rispetto a quello appena descritto per lo stato del robot. Ciò che cambia è la tipologia di evento ed il formato del messaggio infatti il QActor resourcemodel emetterà un evento del tipo `"modelContent", "content( roomMap( state('$content')))` dopo aver ricevuto dal QActor robotmind `forward resourcemodel -m modelUpdate : modelUpdate( roomMap, $Map )`

L'architettura del sistema è riportata in fig. 4.

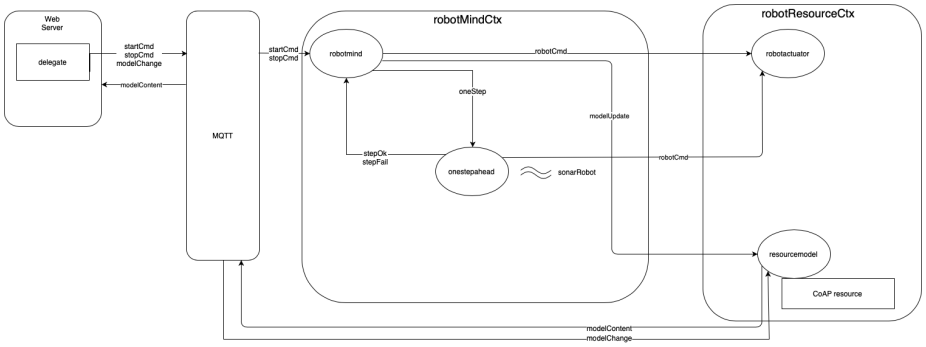

Figure 4: L'architettura del sistema.

4.4 Test plan

Verificare il corretto funzionamento del sistema:

- alla pressione del tasto start/stop sulla web page il sistema parte/si ferma e lo stato della risorsa CoAP cambia coerentemente;
- le informazioni visualizzate sulla web page corrispondano allo stato reale del sistema;

5 Sprint 4

Il progetto di riferimento per questo sprint è it.unibo.ddrSystem5.

OBIETTIVO: creazione di un sistema con un robot che, data una stanza con ostacoli (sia muri sia valigie), la esplori in maniera autonoma e ne costruisca una mappa. Sulla mappa dovranno essere evidenziati tutti gli ostacoli individuati.

5.1 Work Plan

1. definizione di valigia come nuova entità del sistema;
2. definizione di "fine esplorazione" di una stanza con ostacoli fissi (muri e valigie);
3. creazione di una strategia per la gestione degli ostacoli;
4. creazione di una strategia per terminare l'esplorazione di una stanza con ostacoli.

5.2 Analisi dei requisiti

Che cosa si intende per valigia?
La valigia rappresenta un ostacolo fisso disposto all'interno della stanza. Essa può trovarsi in mezzo alla stanza oppure adiacente ad 1 o 2 pareti. Una valigia è un ostacolo che può essere aggirato dal robot, quindi non sarà possibile per il robot esplorare la cella occupata della valigia, ma solo le celle adiacenti. Nel caso in cui un ostacolo si trovi parzialmente su una cella, essa viene comunque considerata dal robot interamente occupata da un ostacolo. Inoltre, in questo sistema non si tiene conto della differenza tra un muro ed una fila lunga di valigie: entrambi determinano il confine della stanza. Perciò, in presenza di un ostacolo, se il robot riesce a raggiungere con un altro percorso il proprio obiettivo, allora continuerà l'esplorazione, in caso negativo significherà che il robot ha individuato i confini.

Che cosa si intende per "fine esplorazione"?
L'esplorazione si considera terminata dopo che il robot ha individuato i confini della stanza, esplorato tutte le celle segnalate con uno 0 sulla mappa ed è tornato nella sua posizione iniziale (0,0).

5.3 Analisi del probelma

La problematica principale che emerge è la gestione degli ostacoli della stanza. In questo sprint sono state individuate due possibili strategie che prevedono l'utilizzo di un sonar posizionato sulla parte frontale del robot. Tali strategie sono:

- il robot, in presenza di un ostacolo prova ad aggirarlo andando alla sua sinistra (questo processo può essere iterato, nel caso in cui un ostacolo

occupi più di una cella). Se il robot ci riesce, considera l'ostacolo una valigia, altrimenti un muro. In figura fig. 5 sono riportate le varie casistiche in cui il robot si potrebbe trovare.

- il robot, in presenza di un ostacolo, calcola un percorso alternativo per raggiungere l'obiettivo. Nel caso in cui non vi siano percorsi disponibili, significa che sono stati individuati i confini della stanza. Altrimenti significa che il robot ha individuato una valigia in mezzo alla stanza o adiacente ad una parete.

Analizzate le due possibili soluzioni si è giunti alla conclusione che la logica applicativa della seconda strategia fosse la più semplice tra le due in quanto permette al robot di gestire nella maniera più opportuna gli ostacoli presenti. Inoltre, si è ritenuto che questa strategia rappresenti una soluzione più elegante in quanto permette di non stravolgere il sistema di partenza.

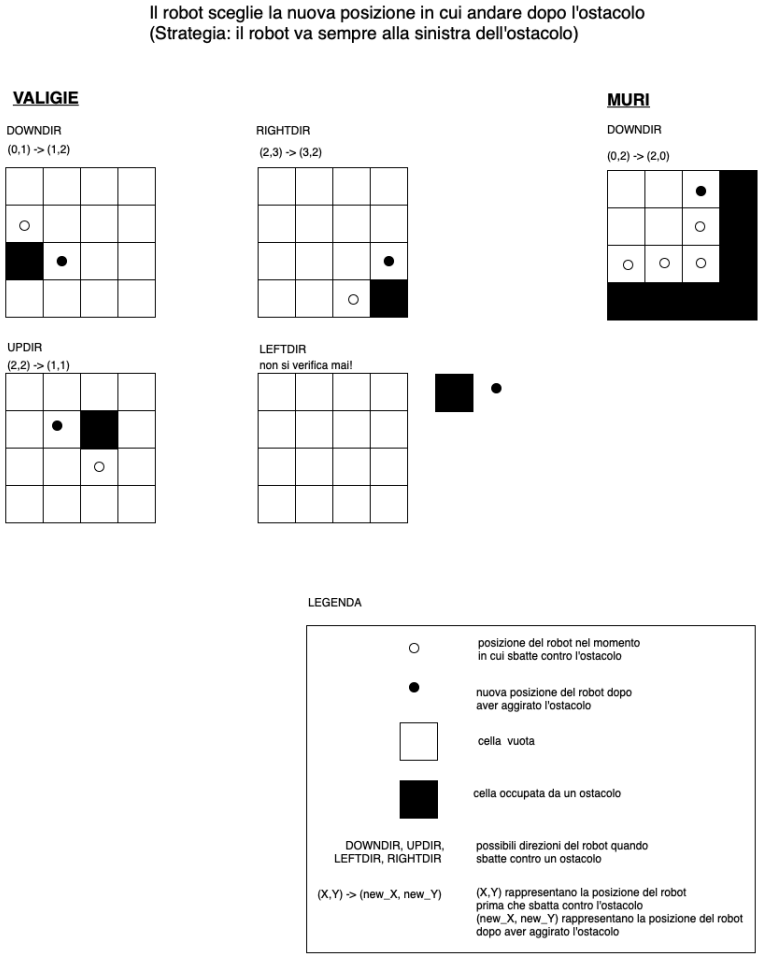

Figure 5: Le varie casistiche che il robot potrebbe dover affrontare.

Un'altra problematica riguarda la gestione della fase di "fine esplorazione" in quanto, una volta che il robot ha individuato i confini della stanza, potrebbero comunque essere rimaste delle celle inesplorate (riportate con uno 0 sulla mappa). Per risolvere questo problema è necessario che il robot le visiti una ad una.

5.4 Model

La logica di funzionamento di robotmind dovrà gestire la nuova tipologia di ostacolo (valigia). Partendo dal presupposto che il robot ispezionerà la stanza utilizzando la strategia della chiocciola (già implementata nello Sprint 2), la nuova logica applicativa del robot dovrà eseguire i seguenti passi:

1. ogni volta che il robot sbatte contro un ostacolo, quest'ultimo viene segnato sulla mappa con una X (già implementata nello Sprint 2);

2. il robot elabora un nuovo piano per provare a raggiungere il goal prefissato, se il piano fallisce ne calcola uno nuovo;

3. quando il robot esaurisce i piani disponibili per raggiungere il goal, significa che le coordinate del goal non rientrano nelle dimensioni della stanza e che entrambe i muri (muro lato sud e lato est) della stanza sono stati individuati;

4. in ultimo, il robot finisce di controllare le celle della stanza rimaste inesplorate (rappresentate con uno 0 sulla mappa). Per ogni cella ancora inesplorata, il robot verifica se la cella è vuota oppure occupata da un ostacolo.

Inoltre, la logica applicativa che riguarda la realizzazione effettiva del piano (doPlan) ed il suo successo o fallimento è stata estrapolata da robotmind ed incapsulata in un nuovo attore (planexecutor), che dato un goal (es. il robot deve raggiungere la posizione (2,2) nella stanza), verifichi se il goal può essere portato a termine e notifichi l'attore (robotmind) del suo successo o fallimento. Il codice di planexecutor è riportato nel Listing 4. Il Listing 5 contiene i messaggi sui quali starà in attesa robotmind a fronte della nuova spartizione dei compiti tra lui stesso e planexecutor. Per quanto riguarda la gestione degli eventi del sonar si è ritenuto opportuno creare un QActor dedicato, denominato sonahandler. Questo QActor, ogni volta che cattura un evento sonarRobot, non invia alcun messaggio nel caso in cui il valore sia maggiore di una certa soglia, poichè significa che non è presente un ostacolo di fronte al robot, in caso contrario invia un messaggio all'attore onestepahead.

```
1
2  QActor planexecutor context robotMindCtx {
3    [”
4      var Curmove      = \”\”
5      var Map = \”\”
6      var Tback = 0
7      var StepTime    = 330
8      //var StepTime    = 700 //fisico
9
10     ”]
11   State s0 initial {}
12
13   Transition t0  whenMsg doPlan  -> loadPlan
14
15   State loadPlan {
16     printCurrentMessage
17     run itunibo.planner.moveUtils.doPlan( myself ) //moves stored
       in actor kb
18   }
19
20   Goto doPlan
21
```

```
22  State doPlan {  }
23
24  Transition t1 whenTime 50  -> doPlan1
25             whenMsg stopCmd -> stopAppl
26
27  State stopAppl {
28    forward robotactuator -m robotCmd : robotCmd(h)
29    forward resourcemodel -m modelUpdate  : modelUpdate( robot, h )
30    solve(retractall( move(_)))
31  }
32
33  Goto s0
34
35  State doPlan1{
36
37    ["Map =  itunibo.planner.plannerUtil.getMapOneLine()"]
38    forward resourcemodel -m modelUpdate  : modelUpdate( roomMap,
      $Map )
39    run itunibo.planner.plannerUtil.showMap()
40
41    solve( retract( move(M) ) )   //consume a move
42    ifSolved {  ["Curmove = getCurSol(\"M\").toString()"]  }
43    else { ["Curmove=\"nomove\" "]  }
44
45  }
46
47  Goto handlemove if "(Curmove != \"nomove\")" else planOk
48
49  State planOk {
50    forward robotactuator -m robotCmd : robotCmd(h)
51    forward resourcemodel -m modelUpdate  : modelUpdate( robot, h )
52    forward robotmind -m planOk : planOk
53  }
54
55  Goto s0
56
57  State handlemove {}
58
59  Goto domove if "(Curmove != \"w\")" else attempttogoahead
60
61  State domove {
62
63    run itunibo.planner.moveUtils.doPlannedMove(myself, Curmove)
64    forward robotactuator -m robotCmd : robotCmd($Curmove)
65    delay 500 //fisico
66    forward robotactuator -m robotCmd : robotCmd(h)
67
68    forward resourcemodel -m modelUpdate  : modelUpdate( robot,
      $Curmove )
69  }
70
71  Goto doPlan
72
73  //roomboundaryplanning.qak
74  State attempttogoahead {
75    forward resourcemodel -m modelUpdate  : modelUpdate( robot, w )
76    run itunibo.planner.moveUtils.attemptTomoveAhead(myself,
```

```
      StepTime)
77  }
78
79  Transition t2   whenMsg stepOk   -> stepDone
80          whenMsg stepFail -> stepFailed
81
82  State stepDone{
83    forward resourcemodel -m modelUpdate  : modelUpdate( robot, h )
84    run itunibo.planner.moveUtils.doPlannedMove(myself, "w")
85
86  }
87
88  Goto doPlan
89
90  State stepFailed{
91    println("&&&  OBSTACLE FOUND")
92    ["var TbackLong = 0L"]
93
94    //printCurrentMessage
95    onMsg( stepFail : stepFail(Obs, Time) ) {
96      ["Tback= (payloadArg(1).toLong().toString().toInt()*0.85 ).
      toInt()
97      TbackLong = Tback.toLong()"]
98    }
99
100   println(" backToCompensate stepTime=$Tback")
101   forward resourcemodel -m modelUpdate  : modelUpdate( robot, s )
102   forward robotactuator -m robotCmd : robotCmd(s)
103
104   delayVar TbackLong
105
106   forward robotactuator -m robotCmd : robotCmd(h)
107   forward resourcemodel -m modelUpdate  : modelUpdate( robot, h )
108
109   forward robotmind -m planFail : planFail
110   solve(retractall( move(_)))
111 }
112 Goto s0
113 }
```

Listing 4: "Codice di planexecutor in ddrSystem5"

```
1
2
3
4  QActor robotmind context robotMindCtx {
5
6     ...
7
8   State startExploration {
9     println("&&&  exploration STARTED")
10
11    run itunibo.planner.plannerUtil.setGoal(X,Y)
12    forward planexecutor -m doPlan : doPlan($X,$Y)
13  }
14
15  Transition t1 ...
16                whenMsg planOk -> nextGoal
```

```
17              whenMsg planFail -> checkIfObstacle
18
19    State nextGoal {
20      if "backHome" {
21        ["
22        backHome = false
23        X = 0
24        Y = 0
25        iterCounter++"]
26      }
27      else {
28      ["
29        backHome = true
30        X = iterCounter
31        Y = iterCounter
32      "]
33      }
34    }
35
36    ...
37
38    State checkIfObstacle {
39      println("--CheckIfObstacle--")
40      run itunibo.planner.moveUtils.setObstacleOnCurrentDirection(
        myself)
41      run itunibo.planner.plannerUtil.resetGoal(X,Y)
42      run itunibo.planner.moveUtils.setObstacleOnCurrentDirection(
        myself)
43      ["plan = itunibo.planner.plannerUtil.doPlan()"]
44
45    }
46
47      ...
48  }
```

Listing 5: "Codice di robotmind in ddrSystem5"

L'architettura del sistema è riportata in fig. 6.

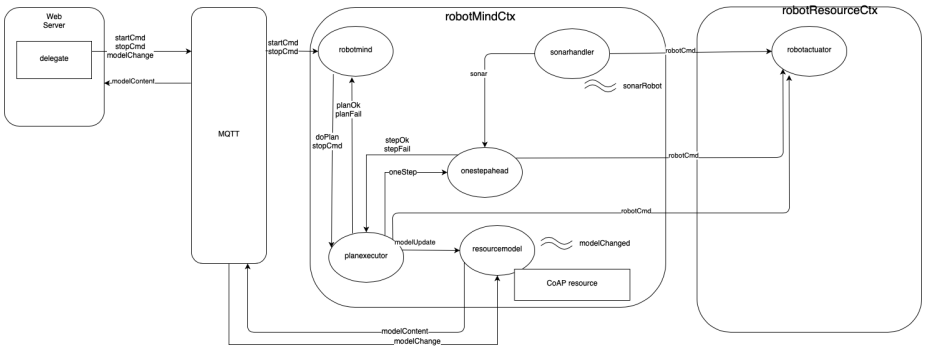

Figure 6: L'architettura del sistema.

5.5 Test plan

Verificare il corretto funzionamento del sistema:

- durante la fase di esplorazione, quando il robot incontra un ostacolo, verificare che l'ostacolo venga segnalato correttamente sulla mappa;
- una volta incontrato un ostacolo, verificare che il robot calcoli un nuovo piano per raggiungere il suo obiettivo;
- verificare che, nel caso in cui l'obiettivo si trovi in corrispondenza di un ostacolo, il robot scelga il goal successivo senza bloccarsi;
- verificare che, nel caso in cui un obbiettivo si trovi all'esterno dei confini il robot termini l'esplorazione;
- verificare che, una volta determinati i confini della stanza, il robot esplori tutte le celle non ancora visitate;

6 Sprint 5

Il progetto di riferimento per questo sprint è it.unibo.ddrsystem6.

OBIETTIVO: creazione di un sistema con un robot che, una volta ricevuto il comando di start (R-starExplore), inizi a esplorare autonomamente la stanza in cui si trova (R-explore). Ogni volta che il robot incontra un ostacolo dovrà fermarsi(R-stopAtBag), scattargli una foto (R-takePhoto), inviarla alla console dell'operatore (R-sendPhoto) ed aspettare l'esito della verifica. In caso in cui la valigia risulti sospetta (ovvero contenete la bomba) il robot dovrà memorizzare la posizione della valigia (R-storePhoto), tornare alla propria base (R-backHomeSincePhoto). In caso contrario dovrà continuare l'esplorazione (R-continueExploreAfterPhoto).

6.1 Work Plan

1. definizione di valigia sospetta;
2. definizione di "base del robot";
3. gestione dell'invio della foto di ogni ostacolo;
4. gestione della ricezione del messaggio contenente il risultato dell'analisi della bomba.

6.2 Analisi dei requisiti

Cosa si intende per valigia sospetta?
Una valigia la si definisce sospetta quando, attraverso un tool esterno al sistema, si verifica se dentro di essa sia presente o meno una bomba.

Cosa si intende per "base del robot"?
Il concetto di "base del robot" rimane invariato rispetto agli sprint precedenti: essa è il punto da cui il robot parte per esplorare la stanza. Nel caso specifico si è scelto l'angolo in alto a sinistra di una stanza, definito come (0,0).

6.3 Analisi del problema

Come si deve comportare il robot quando incontra un ostacolo?
Ogni volta che il robot incontra un ostacolo effettua un passo indietro per tornare nella cella precedente (backToCompensate). Tterminata questa fase, il robot si ferma ("R-stopAtBag"), scatta una foto ("R-takePhoto") ed infine la invia ("R-sendPhoto") al device dell'operatore. Infine il robot si mette in attesa di una risposta ("luggageSafe" or "luggageDanger").

Cosa succede se la risposta ("luggageSafe" oppure "luggageDanger") non arriva mai al robot?
Nel caso in cui il robot non riceva la risposta dalla console rimane fermo nella sua posizione.

Cosa fa quando riceve il messaggio di "luggageDanger"?
Quando riceve questo messaggio, significa che l'ultima valigia esaminata è quella contenete la bomba, dunque il robot dovrà tornare alla base ("R-backHomeSinceBomb").

Cosa fa quando riceve il messaggio di "luggageSafe"?
Quando riceve questo messaggio, significa che l'ultima valigia esaminata non contiene la bomba, dunque il robot dovrà continuare l'esplorazione e la verifica delle altre valigie ("R-continueExploreAfterPhoto").

Come faccio a mostrare ed aggiornare le informazioni della valigia sulla web page? Per la visualizzazione dello stato della valigia corrente si è deciso di utilizzare le informazioni gestite dalle plannerUtils. La risorsa verrà rappresentata sulla web page nel seguente modo: `luggage(photo_id, position(X,Y), date_time)`
Dove:

- photo_id rappresenta la foto della valigia scatta dal robot;
- position(X,Y) le coordinate della valigia nella mappa del robot;
- la data e l'ora in cui è stata scattata la foto;

Lo stato dell'ultima valigia mostrato sulla web page sarà quello relativo alla valigia con la bomba.

6.4 Model

Il committente richiede di salvare le informazioni della valigia sospetta (immagine, posizione, ora). Dato che inizialmente non si sa quale sia la bomba e la sua posizione, si ritiene importante salvare le informazioni solamente della valigia da ispezionare attualmente e:

- nel caso in cui la verifica risulti negativa alla verifica da parte del tool le informazioni verranno sovrascritte da quelle della valigia seguente;
- altrimenti verranno salvate esattamente quelle della valigia sospetta.

Queste informazioni si potrebbero tener dentro al sistema (ad esempio per motivi di sicurezza o efficienza) oppure si potrebbero rendere accessibili da altri osservatori utilizzando la stessa strategia utilizzata per lo stato del robot e della stanza. Quest'ultima strategia potrebbe essere più efficace in quanto, essendo il tool utilizzato per la verifica esterno al sistema, potrebbe accedere a tali informazioni attraverso uno standard (CoAP) in maniera più trasparente che un messaggio "custom" della SoftwareFactory QKActor. Allo stesso tempo la risorsa potrebbe (la valigia) avere un attributo chiamato "safe" settato di default a "true" e a "false" dal tool di verifica. Dunque anche il nostro sistema dovrà essere un observer di tale risorsa in quanto, quando viene cambiato questo attributo, il sistema deve eseguire tutte le azioni descritte in precedenza. Tuttavia tale soluzione risulta superflua, dato che, è richiesto il salvataggio solo della valigia sospetta. Il Listing 6 contiene la logica di gestione degli ostacoli della robtmind.

```
1
2  QActor robotmind context robotMindCtx {
3
4  ...
5
6      State newLuggageFound {
7        //println("========== robotmind: newLuggageFound ==========")
8        ["Luggage_num++"]
9        forward resourcemodel -m modelUpdate: modelUpdate(luggage,
       $Luggage_num)
10
11       run itunibo.planner.moveUtils.setObstacleOnCurrentDirection(
       myself)
12       ["Map =  itunibo.planner.plannerUtil.getMapOneLine()"]
13       forward resourcemodel -m modelUpdate:modelUpdate(roomMap,
       $Map)
14
15
16
17     }
18     Transition t2 whenMsg luggageSafe -> handleObstacle
19             whenMsg luggageDanger -> endExploration
20
21
22     State handleObstacle {
23       //println("========== robotmind: handleObstacle ==========")
24       run itunibo.planner.moveUtils.setObstacleOnCurrentDirection(
       myself)
25       run itunibo.planner.plannerUtil.resetGoal(X,Y)
26       run itunibo.planner.moveUtils.setObstacleOnCurrentDirection(
       myself)
27       ["plan = itunibo.planner.plannerUtil.doPlan()"]
28     }
29     Goto startExploration if "(plan != null)" else checkNull
30 }
```

Listing 6: "Codice di robotmind in ddrSystem6"

Ogni qualvolta la console viene notificata della presenza di un ostacolo, sulla web page si attivano due bottoni ("safe" e "dangerous") che permettono all'operatore di segnalare al sistema se l'ostacolo è pericoloso oppure no. Dopo che l'operatore ha premuto uno dei due bottoni, entrambi si disattivano fino all'ostacolo successivo. Il Listing 7 contiene la PUT fatte alla risorsa CoAP nel progetto frontend (classe applCode.js).

```
1
2
3  app.post("/danger", function(req, res,next) { handlePostMove("
       danger","going to initial position", req,res,next); });
4  app.post("/safe", function(req, res,next) { handlePostMove("safe","
       continuing the exploration", req,res,next); });
```

Listing 7: "Put alla risorsa CoAP nel progetto frontend"

L'architettura del sistema rimane per lo più invariata rispetto allo sprint precedente. Gli unici messaggi che vi sono in più sono quelli utilizzati per

segnalare la presenza di ostacolo pericoloso o meno (lato web page, tali messaggi corrispondono a delle PUT, con "safe" o "danger", alla risorsa CoAP).

6.5 Test plan

Verificare il corretto funzionamento del sistema:

- verificare che il robot, una volta incontrato un ostacolo, si metta in attesa del messaggio dell'operatore;
- verificare che il robot, una volta ricevuto il messaggio di "luggageSafe", continui l'esplorazione della stanza;
- verificare che il robot, una volta ricevuto il messaggio di "luggageDanger", torni alla base terminando la fase di esplorazione e senza verificare le celle non ancora visitate.

7 Sprint 6

Il progetto di riferimento per questo sprint è it.unibo.ddrsystem6.

OBIETTIVO: creazione di un sistema con un robot che una volta ricevuto il comando di "start", inizi a esplorare autonomamente la stanza in cui si trova solo se la temperatura della stanza è al di sotto di una certa soglia (R-tempOk). Una volta iniziata la fase di esplorazione, non appena la temperatura della stanza supera la soglia il robot si deve fermare. Durante l'esplorazione, se il robot riceve il comando di backHome (R-backHome), il robot deve sospendere l'esplorazione e tornare alla base. Durante l'esplorazione il robot deve anche far blinkare un led (R-blinkLed).

Work plan:

1. definizione di una strategia con cui poter controllare la temperatura della stanza;
2. gestione del cambio di temperatura nel caso in cui essa superi una certa soglia;
3. gestione del messaggio di "backHome";
4. gestione del blinking del led posto sul robot durante la fase di esplorazione.

7.1 Analisi dei requisiti

Cosa si intende per variazione della temperatura della stanza? Per variazione della temperatura della stanza si intende un cambiamento significativo di essa che la porti ad assumere un valore superiore o inferiore ad una soglia fissata.

Cosa si intende per blinking del led?
Per blinking del led si intende, nel caso del robot fisico, l'accensione ad intermittenza un led posto su di esso. Nel caso del robot simulato tale comportamento è simulato con una stampa su console ("blinking").

7.2 Analisi del problema

Come è possibile gestire la variazione della temperatura della stanza?

La variazione è gestita utilizzando due bottoni "OK" e "TOOHIGH". Quando il primo viene premuto significa che la temperatura è al di sotto di una certa soglia e non sono state rilevate variazioni particolarmente significative (temperatura accettabile). Invece, quando viene premuto il secondo bottone significa che vi è stata una variazione della temperatura tale da comportare l'arresto dell'esplorazione (temperatura alta). Ad inizio esplorazione si ipotizza che la temperatura della stanza sia corretta. Una volta che il bottone "TOOHIGH" viene premuto, il robot deve sospendersi in attesa che la temperatura ritorni nella norma (premendo il bottone "OK") e che l'operatore ridia il comando di "start".

Come è possibile gestire il comando di "backHome"?
L'esplorazione del robot è suddivisa nel raggiungimento di diversi goal. (i.e. (0,0)(1,1)(2,2)...) Quando l'operatore spinge il bottone di "backHome" è sufficiente indicare al robot che il goal da perseguire corrisponde alla propria base (0,0) e una volta che lo ha raggiunto aspettare che l'operatore ridia il comando di "start" per riprendere l'esplorazione.

Come è possibile gestire il blinking del led?
Il task del blinking, pur essendo una funzionalità a parte rispetto a quella di esplorazione, deve comunque essere gestito in contemporanea rispetto a quest'ultimo. Quindi potrebbe risultare utile incapsulare tale comportamento in un'entità separata

7.3 Model

Per quanto riguarda la gestione della temperatura, è necessario modificare il behaviour del QKActor robotmind. Robotmind, sia prima di iniziare l'esplorazione, sia mentre questa è in corso, controlla se nel sistema si verifica un evento di "temperatureTooHigh". Nel primo caso, se ciò accade, il robot dovrà aspettare che si verifichi l'evento di "temperatureOk" e un comando di "start" prima di dare il via all'esplorazione. Nel secondo caso, il robot, arresta momentaneamente l'esplorazione sospendendo il piano che stava svolgendo, ponendosi prima in attesa dell'evento "temperatureOk" e poi del comando di "start" che gli consenta di riprenderla. Nel caso l'evento di "temperatureOk" non arrivi mai, il robot rimarrà fermo nella sua posizione attuale. Il listing 8 riporta il codice per gestire la logica applicativa della temperatura.

```
1  Event temperatureTooHigh: temperatureTooHigh
2
3  QActor robotmind context robotMindCtx {
4
5    State waitForStart {
6      ...
7    }
8
9    Transition t0   whenEvent startCmd  -> startExploration
10            whenEvent temperatureTooHigh  -> waitForTemperatureOk
11
12     ...
13
14     State startExploration {
15       ...
16
17   }
18
19   Transition t1 whenEvent temperatureTooHigh ->
       waitForTemperatureOk
20                 ...
21
22     State backHome {
23        println("========== robotmind: backHome ==========")
24       forward planexecutor -m stopPlan: stopPlan
25       run itunibo.planner.plannerUtil.setGoal(0,0)
```

```
26          forward planexecutor -m doPlan : doPlan(0,0)
27      }
28      Transition t2 whenEvent temperatureTooHigh  ->
          waitForTemperatureOk
29                      ...
30
31      State handleStartAfterBackHome { }
32      Transition t0  whenEvent temperatureTooHigh  ->
          waitForTemperatureOk
33                      ...
```

Listing 8: Codice di QActor robotmind in ddrSystem6

Quando l'operatore clicca il pulsante "backHome" sulla propria console tale evento viene catturato dal QKActor robotmind quando quest'ultimo si trova nello stato "startExploration". Una volta catturato tale evento prima di tutto interrompe il piano attualmente in esecuzione e poi imposta il nuovo goal a (0,0) in modo da tornare alla base. Una volta terminato il goal, robotmind, si mette in attesa dello "start". Nel caso in cui, durante tutto il procedimento, dovesse percepire l'evento di "stopCmd" oppure di "temperatureTooHigh", sospenderà l'esecuzione del compito per attendere gli opportuni comandi prima di ricominciare. Il listing 9 riporta il codice per gestire la logica applicativa del backHome.

```
1
2  State startExploration {... }
3
4    Transition t1 ...
5                 whenEvent backHomeCmd -> backHome
6
7    State backHome {
8        forward planexecutor -m stopPlan: stopPlan
9        run itunibo.planner.plannerUtil.setGoal(0,0)
10       forward planexecutor -m doPlan : doPlan(0,0)
11   }
12   Transition t2 whenMsg planOk -> handleStartAfterBackHome
13              whenMsg planFail -> backHome
14              whenEvent stopCmd -> waitForStart
15              whenEvent temperatureTooHigh  -> waitForTemperatureOk
16
17   State handleStartAfterBackHome {  }
18   Transition t0  whenEvent startCmd  -> startExploration
19            whenEvent temperatureTooHigh  -> waitForTemperatureOk
```

Listing 9: "Codice di robotmind per la gestione del backHome"

Dal momento che si è individuato il compito del blinking del led come task separato rispetto a quello dell'esplorazione, ma che deve comunque essere svolto contemporaneamente, si è deciso di assegnare la gestione di tale compito ad un QKActor diverso da robotmind, denominato blinkinghandler. Tale attore, ogni volta che riceve il messaggio di "startBlinking" dovrà comunicare al robotactuator di iniziare il compito del blinking, e quando riceve il messaggio di stopBlinking, dovrà comunicare al robotactuator di terminarlo. Dati i requisiti, la robotmind indicherà al blinkinghandler di avviare tale task non appena inizia la fase di esplorazione e di terminarlo quando viene individuata la valigia

con la bomba. Il listing 10 riporta il codice per gestire la logica applicativa del blinkingLed.

```
1
2  QActor blinkinghandler context robotMindCtx {
3
4      State s0 initial {     }
5      Transition t0 whenMsg startBlinking -> sendBlinkingMsg
6
7      State sendBlinkingMsg{
8        println("========== blinking ==========")
9        forward robotactuator -m robotCmd : robotCmd (blinking)
10     }
11     Transition t1 whenMsg stopBlinking -> stopBlinking
12
13     State stopBlinking{
14       println("========== stop blinking ==========")
15       forward robotactuator -m robotCmd : robotCmd (stopBlinking)
16     }
17     Goto s0
18 }
```

Listing 10: "Codice di blinkinghandler in ddrSystem6"

In questo sistema si è revisionata anche la strategia di esplorazione: ora il robot, terminata una "chiocciola", prima di passare a quella successiva, esplora tutte le caselle a "0" presenti, senza esplorarle tutte una volta trovate le pareti. Questa nuova strategia permette di trovare un'eventuale bomba "as soon as possible".

L'architettura del sistema è riportata in fig. 7.

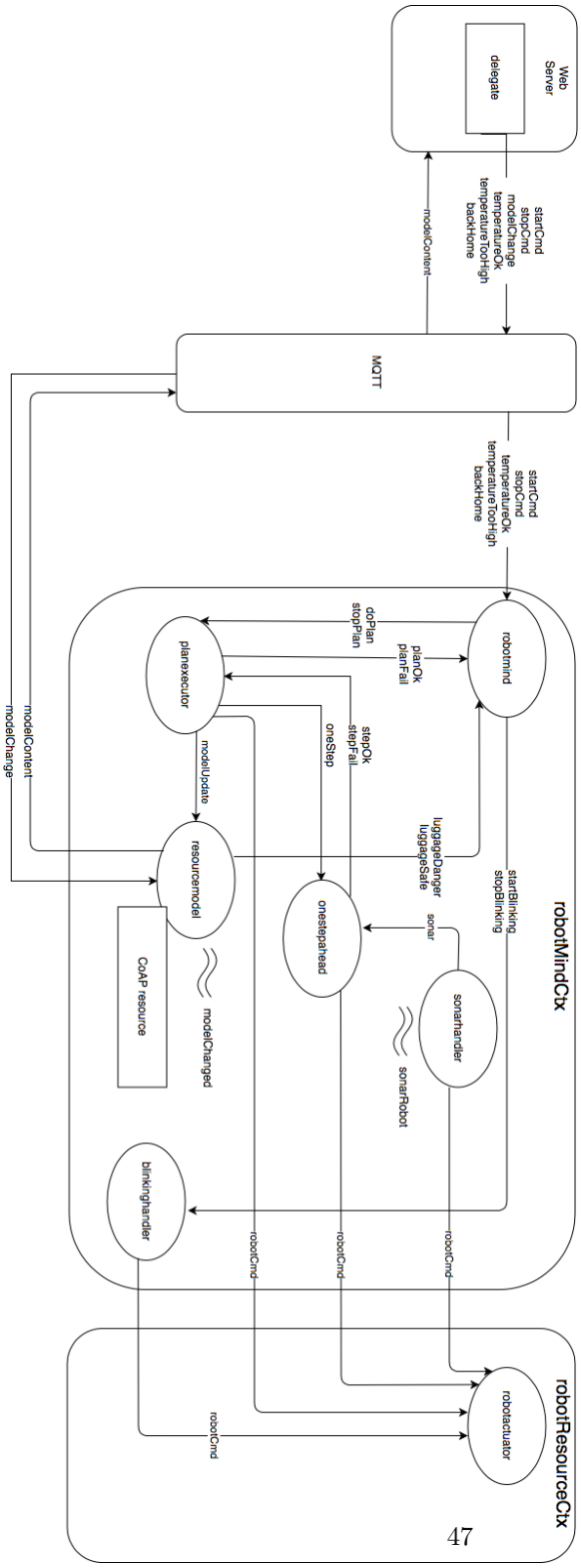

Figure 7: L'architettura del sistema.

7.4 Test plan

Verificare il corretto funzionamento del sistema:

- verificare che il robot, una volta percepito l'evento di "temperatureTooHigh", sospenda l'esplorazione;
- verificare che il robot, una volta sospesa l'esplorazione a seguito dell'evento "temperatureTooHigh" la riprenda solo una volta percepiti in ordine gli eventi "temperatureOk" e "startCmd":
- verificare che il robot, una volta ripresa l'esplorazione dopo averla interrotta a causa della temperatura oltre la soglia, reimposti il goal corretto, ossia quello che stava svolgendo prima di fermarsi;
- verificare che il robot, una volta percepito l'evento di "backHome", imposti il goal a (0,0);
- verificare che il robot, una volta tornato alla base poiché ricevuto il comando di "backHome", si metta in attesa del comando di "start";
- verificare che il robot, una volta ripresa la fase di esplorazione, dopo averla interrotta a causa del comando di "backHome", reimposti il goal corretto: quello che stava svolgendo prima di impostare (0,0);
- verificare che il robot, non appena inizia la fase di esplorazione, avvi anche il blinking del led sul robot;
- verificare che il robot, non appena termina la fase di esplorazione (o perché ha trovato la bomba o perché ha esplorato tutte le celle della stanza), termini anche il compito del blinking del led;

Sprint0 Review

30/12/2018
In questo sprint si sono definiti in modo formale i requisiti del sistema e si è analizzato in dettaglio il problema. Dall'analisi del problema è emersa l'architettura logica del sistema che è stata formalizzata utilizzando i QActor.

Sprint1 Review

27/06/2019
In questo sprint si è modificato il sistema dello sprint0 in modo da permettere al robot di gestire (ricevere ed eseguire) i comandi base di spostamento: "vai avanti", "vai indietro", "gira a destra", "gira a sinistra", "fermati". Ogni volta che esegue un comanda il sistema modifica coerentemente la sua base di conoscenza.

Sprint2 Review

15/07/2019
In questo sprint si è modificato il sistema dello sprint1 in modo da permettere al robot (fisico e virtuale) di esplorare una stanza rettangolare e vuota. Durante l'esplorazione il robot costruisce internamente una rappresentazione della stanza (utilizzando le plannerUtils). Essendo la stanza vuota, l'unico ostacolo modellato in questo sistema sono i muri. Quando il robot riconosce un muro (attraverso l'uso di un sonar), esso fa un passo indietro, si ferma, salva l'informazione sulla mappa e procede con il goal seguente.

Sprint3 Review

11/08/2019
In questo sprint si è modificato il sistema dello sprint2 in modo da permettere al robot virtuale di partire con l'esplorazione automatizzata della stanza quando questa viene azionata sulla web page tramite il bottone start. Inoltre, si è creata una risorsa CoAP che rappresenti lo stato del robot e della stanza, il quale sarà osservabile direttamente sulla web page e aggiornato ogni qualvolta si modifichi.

Sprint4 Review

29/08/2019
In questo sprint si è modificato il sistema dello sprint3 in modo da permettere al robot virtuale di esplorare autonomamente una stanza che contiene ostacoli fissi diversi dai muri (valigie). Questo ha richiesto di effettuare il refactoring dei diversi task assegnati precedentemente i QKActor. In particolare si è incapsulata in un nuovo attore (planexecutor) la gestione dell'attuazione di singolo un piano, e si è esteso il comportamento di robotmind modificando la logica applicativa del nuovo sistema.

Sprint5 Review

4/09/2019
In questo sprint si è modificato il sistema dello sprint4 in modo da permettere al robot virtuale di esplorare autonomamente una stanza che contiene ostacoli fissi diversi dai muri (valigie). In questo sistema, ogni volta che il robot incontra un ostacolo, si sospende in attesa dell'esito della verifica, con la relativa gestione di entrambe le casistiche. Questo ha richiesto di modificare la logica applicativa incapsulata nel comportamento di robotmind: tale attore ora starà in attesa di ricevere due possibili messaggi: "luggageSafe" o "luggageDanger".

Sprint6 Review

12/09/2019
In questo sprint si è modificato il sistema dello sprint5 in modo da permettere al robot virtuale di esplorare autonomamente una stanza che contiene ostacoli fissi. In questo sistema, ogni volta che la temperatura della stanza diventa più alta di una soglia, il robot si ferma in attesa che la temperatura si abbassi e che l'operatore gli ridia il comando di "start". Durante l'esplorazione, quando l'operatore manda il comando di "backHome", il robot sospende il goal corrente per tornare alla base e attendere di nuovo il comando di "start", così da riprende l'esplorazione da dove l'aveva lasciata. Durante tutta la fase di esplorazione il robot esegue anche un altro compito: far blinkare un led posto su di esso. In questo sistema si è modificata la logica applicativa incapsulata nel comportamento di robotmind: tale attore ora potrà percepire anche gli eventi di: "temperatureTooHigh", "temperatureOk" e "backHome". Si è aggiunto un attore blinkinghandler per la gestione del blinking del led: quest'ultimo comincia la fase di blinking non appena riceve il messaggio di "startBlinking" e la termina quando riceve quello di "stopBlinking". "startBlinking" e "stopBlinking" sono entrambi messaggi inviati da robotmind.