Robot Discovery - ddr

Elaborato di progetto per il corso Sviluppo di Sistemi Software

`https://bitbucket.org/chiara-volonnino/iss-18-ddr`

Chiara Volonnino[1]

Email ids: [1]chiara.volonnino@studio.unibo.it

A.A. 2018/2019

Indice

1 Sprint 1 **3**
- 1.1 Requisiti . . . 3
- 1.2 Analisi dei requisiti . . . 4
- 1.3 Analisi del problema . . . 5

2 Sprint 2 **7**
- 2.1 Word Observer . . . 7
- 2.2 Mock del sistema . . . 7
- 2.3 Introduzione ai QActor . . . 8
- 2.4 Una prima architettura logica . . . 8
  - 2.4.1 Finite State Automaton (FSA) . . . 8
- 2.5 Prima implementazione . . . 9

3 Sprint 3 **12**
- 3.1 Planner . . . 12
  - 3.1.1 Approccio: divisione dello spazio in celle . . . 12
  - 3.1.2 Approccio: unità di misura, il tempo . . . 12
  - 3.1.3 Approccio: robot-planner . . . 13
- 3.2 Prima implementazione . . . 14
  - 3.2.1 Approccio: one cell forward . . . 14

4 Sprint 4 **16**
- 4.1 Console . . . 16
  - 4.1.1 Considerazioni . . . 16
  - 4.1.2 Tecnologie . . . 16
- 4.2 Canali di comunicazione . . . 17
  - 4.2.1 WebSocket . . . 17
  - 4.2.2 MQTT: Message Queue Telemetry Transport . . . 17
- 4.3 Temperatura . . . 17
  - 4.3.1 Modifica di requisito . . . 17
- 4.4 Interfaccia grafica . . . 17

5 Sprint 5 **18**
- 5.1 Fotografia della borsa . . . 18
  - 5.1.1 Problema: dotazione della fotocamera . . . 18
- 5.2 Informazioni relative allo stato . . . 18
- 5.3 Storage delle foto . . . 19
- 5.4 Robot retriever . . . 19
  - 5.4.1 Recupero della borsa: sviluppo futuro . . . 19
- 5.5 Gestione della mappa . . . 19
  - 5.5.1 Condivisione della mappa . . . 20
  - 5.5.2 Tipo di comunicazione . . . 20
- 5.6 Un implementazione . . . 20

6 Sprint 6 **23**
6.1 Sistema distribuito 23

7 Il sistema **24**
7.1 Architettura logica 24
7.1.1 Finite State Automaton (FSM) 24
7.1.2 Interazioni 25

8 Tecnologie **26**
8.1 QActor 26
8.2 Robot Virtuale 26
8.3 Robot Fisico 27
8.3.1 Arduino: configurazione 27
8.4 MQTT 28
8.5 Front-end 28
8.5.1 Node 28
8.5.2 Express 28
8.5.3 Angular 28

9 Note **29**
9.1 MQTT 29
9.2 Attori dinamici 29
9.3 Problemi con Windows 29

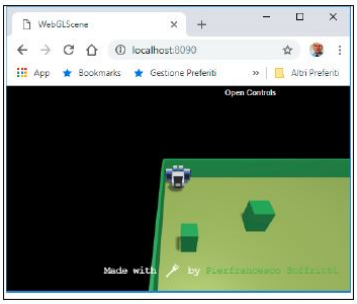
Capitolo 1

Sprint 1

1.1 Requisiti

Un *ddr* robot (discovery) viene usato per verificare l'esistenza di una bomba all'interno della hall di un aeroporto. La suddetta hall deve essere evacuata ma devono rimanere i bagagli sul pavimento. Il discovery robot è controllato in modo remoto da uno smart device pilotato da un operatore umano che opera in una zona protetta. Il robot può partire solo quando sono verificate due condizioni:

- l'operatore umano invia un comando di Explore (R-startExplore) tramite l'ausilio di una interfaccia grafica funzionante sullo smart device;
- il valore della temperatura della hall non è più alta di un valore prefissato (R-tempOk).



Il sistema software che funziona sul robot e sul device dell'operatore deve fornire le suddette funzionalità:

1. il robot deve poter esplorare la hall in modo autonomo (R-explore) con l'obiettivo di raggiungere ogni borsa presente sul pavimento;
2. durante l'esplorazione l'operatore può fermare (R-stopExplore) il robot e inviare comandi per farlo tornare nella sua posizione iniziale (R-backHome) o per fargli continuare la fase esplorativa (R-continueExplore);
3. se il robot scopre qualcosa durante l'esplorazione deve far lampeggiare un led (R-blinkLed) su di lui inserito e inviare/aggiornare i dati riguardanti lo stato della scoperta e lo stato del robot sulla console che possiede l'operatore umano;
4. quando è in prossimità di una borsa il robot deve:
   (a) stopparsi (R-stopAtBag);
   (b) scattare una foto (R-takePhoto) della borsa;
   (c) inviare la foto al device dell'operatore (R-sendPhoto).

5. quando il device dell'operatore riceve la foto esegue un tool per controllare e quindi capire se la borsa contiene effettivamente una bomba. In caso di esito positivo (quindi in caso che la borsa contenga una bomba):

   (a) il device deve notificare la scoperta all'operatore (R-alert);

   (b) memorizzare la foto (R-storePhoto) su un dispositivo di memorizzazione permanente insieme alle informazioni contestuali (es. data e ora, posizione, etc.);

   (c) inviare al robot il comando di tornare alla posizione iniziale (R-backHomeSinceBomb).

   In caso contrario (quindi non si è riscontata una bomba in quella borsa) l'operatore invia al robot il comando per continuare l'esplorazione (R-continueExploreAfterPhoto);

6. l'operatore quando viene allertato per possibile bomba:

   (a) deve aspettare che il robot sia tornato nella sua posizione iniziale (R-waitForHome);

   (b) quando il robot è tornato deve notificare ad un altro robot (dotato di strumenti adeguati) di raggiungere la borsa sospetta (R-reachBag) per mettere la borsa in un posto sicuro e trasportarla fino alla posizione iniziale (R-bagHome).

Obiettivo di business

Implementare un sistema che permetta ad un utente umano di controllare un robot remoto capace, fra le altre, di esplorare l'ambiente in cerca di bombe e che sia capace allo stesso tempo di tracciare dinamicamente una mappa di "navigazione" dell'ambiente a lui circostante.

1.2 Analisi dei requisiti

Dopo una attenta analisi e un'accurata discussione con il committente si sono delineate le seguenti osservazioni.

Il sistema software, dopo una prima revisione, risulta essere un sistema distribuito composto da due entità principali: il **robot** e la **console**. Nello specifico:

- il robot è un entità remota (reale o virtuale) autonoma e reattiva che ricevere comandi da un utente umano attraverso la console; In particolar modo il robot è può essere distinto in:
  - un robot **discovery**, il robot esploratore;
  - un robot **retriever**, il robot che si occuperà di recuperare la bomba.
- la **console** ha il compito di catturare i comandi attuati dall'operatore umano e inviare questi comandi al robot remoto. Questa è un entità autonoma che può essere in esecuzione si un qualsiasi dispositivo Smart-Divice o su un Personal Computer.

Il robot per poter iniziare l'esplorazione deve verificare che alcune condizioni ambientali siano favorevoli alla sua inizializzazione. Nello specifico:

- il robot deve restare in ascolto per ricevere dall'utente il comando di start exploration;
- deve verificare che la **temperatura** nella hall non superi un limite prefissato;

Per controllare la temperatura, si è deciso con il committente di escludere la possibilità di avere un qualsivoglia dispositivo di controllo temperatura sulla console, in quanto rischierebbe di non essere uniforme con la reale temperatura della hall in cui lavorerà il robot, essendo la postazione remota. Inoltre tale punto può essere interpretato come servizio aggiuntivo che offre il nostro sistema.

Il robot è dotato di un **sonar**, posto sul suo fronte, per permettergli di essere sensibile all'ambiente. Questa tecnologia quindi rimane indispensabile per il suo funzionamento; altre tecnologie sarebbero sicuramente di supporto, ma per il momento, dati i costi, si è deciso con il committente di escluderle.

Il robot reale deve include nel suo equipaggiamento dei **led**, indispensabili per notificare all'utente umano che ha scoperto qualcosa e un dispositivo che gli permetta di catturare una **foto** delle borse, in modo da inviarla alla console e permettere un'analisi in grado di intendere se tale borsa contiene o meno una bomba.

Il software sulla console deve permettere all'utente finale di interagire con il robot in modo chiaro e semplice e quindi deve includere dei comandi che permettano tale scopo (es. Start, Stop, BackHome, ContinueExploration). A tal fine il software sul robot deve interpretare i comandi inviati dall'utente tramite la console e inviarli al robot che deve poterli attuare in modo completamente autonomo.

Come già preannunciato il robot deve essere **sensibile all'ambiente**, con particolare riferimento alla temperatura e ai possibili ostacoli presenti sul piano della hall. In questo contesto quindi faremo riferimento a tali condizioni come **enviromentCondition**. Nello specifico delineiamo che: gli ostacoli posti nella hall devono essere rilevati **dinamicamente** dal robot tramite l'ausilio dei sonar; mentre, per quanto concerne la temperatura, deve essere acquisita in qualche modo.

Il robot deve avere la capacità di costruire in modo dinamico una **mappa** del territorio, capace di mostrare anche gli ostacoli incontrati. Quindi risulta fondamentale una **fase di planning** dell'ambiente circostante al robot.

Il robot deve essere, in qualsiasi momento, reattivo al comando di *halt* proveniente dalla console, anche se esso si trova in stato di esplorazione.

Il device console deve inoltre includere un tool che permetta di controllare in qualche modo, per ogni foto ricevuta dal robot, se la borsa contiene o meno una bomba.

Inoltre, se il tool di ricerca bombe produce un esito positivo, il device deve includere un qualche sistema di **alerting** per notificare all'operatore finale la scoperta, e proseguire con il salvataggio della foto e delle relative informazioni (es. data e ora, posizione, etc) su un qualche dispositivo di memorizzazione permanente. Infine, deve notificare al robot di tornare alla posizione di partenza (home).

Infine, a fronte di una scoperta, il robot deve ricevere il comando di ritornare nella sua posizione iniziale e, una volta giunto, deve avere la capacità di notificare in qualche modo, ad un altro robot (**robot retriever**) la posizione della borsa.

Il robot retriever, deve poter raggiungere con facilità la borsa contenente la bomba, quindi deve conoscere il piano di navigazione creato dal primo robot. Una volta raggiunta tale bomba, deve, in qualche modo raccoglierla e portarla in un posto sicuro per disinnescarla.

1.3 Analisi del problema

Dopo aver analizzato e discusso con il committente i requisiti da lui posti, procediamo con l'analisi del problema.

In tale accezione, ci accorgiamo che la logica di business del sistema è principalmente legata al software che si occupa di implementare il comportamento del robot. Diamo così il nome **Robot Discovery** a tale parte che deve essere in grado di creare un robot autonomo capace di perlustrare una stanza e che sia reattivo agli ostacoli e agli eventi.

Risulta inoltre utile dividere il concetto di discovery robot in due parti:

- *Robot-Mind*, che avrà il compito di implementare la strategia necessaria a risolvere l'obiettivo di business (implementerà la **business logic** del sistema software);
- *Robot-Adapter*, che eseguirà i comandi per muovere il robot (attività passiva di semplice attuatore). Quindi si occuperà di eseguire i comandi ricevuti dalla mind, senza conoscere la componente logica.

Quindi il robot-mind averà il compito di monitorare l'ambiente e gli eventi, oppure si può optare di delegare tali compiti ad una terza entità, chiamata **Word-Observer**. Nel nostro contesto per avere una maggiore riusabilità, si é deciso di creare tale entità.

Da subito rimane indispensabile definire un piano di Planning, utile per permettere al robot di: valutare in modo corretto tutto il territorio che lo circonda, indispensabile per catturare e implementare la business core di tale sistema, l'esplorazione; e definire anche i punti in cui sono presenti degli ostacoli. Questa mappa del territorio inoltre, risulta indispensabile per permettere al secondo robot di raggiungere in completa autonomia e semplicità la borsa contenente la bomba e portarla al sicuro.

Quindi d'ora in poi definiremo:

- *Discovery-Robot* il primo robot, ovvero colui che ha il compito di esplorare la hall e valutare le diverse borse in cerca della bomba;
- *Retrieval-Robot* il secondo compito, ovvero colui che ha il compito di andare a prelevare la borsa che ha lanciato l'allarme, seguendo il percorso indicatogli sulla mappa dal discovery-robot.

Definire la mappa del territorio in questi termine ci permette di definire che il sistema deve mantenere informazioni, in quanto tale mappa deve essere trasferita in qualche modo al retrieval-robot.

Quando parte il retrieval-robot dobbiamo assumere che l'ambiente non sia cambiato.

Capitolo 2

Sprint 2

Per questa sprint é stato richiesto dal committente di portare un mock del nostro sistema.

Nello specifico si é deciso di implementare un semplice mock del in cui il robot non aveva comportamenti complessi, l'esplorazione non é stata ancora attivata, e quindi il robot doveva solamente avanzare alla ricezione del comando explore (requisito **R-startExplore**) e fermarsi al comando halt ricevuto da parte dell'operatore (requisito **R-stopExplore**).
In particolare per tale dimostrazione si é utilizzato solo il robot virtuale, in quanto il robot fisico era ancora in fase di assemblaggio.

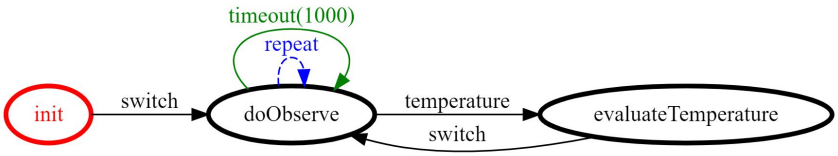
Si é poi creato da subito il **Word Observer**.

In fine si é implementato un primo prototipo per l'interfaccia grafica per poter mostrare al committente un sistema funzionante, in tale accezione si é partito dal sistema offertoci dal nostro committente. .

2.1 Word Observer

Il *Word Observer* é un componente capace di comunicare con il mondo esterno, percependo i cambiamenti che avvengono in questo. In tale modo ci é stato utile per verificare le condizioni ambientali, necessarie per fare avvivare il robot (requisito **R-TempOk**).

Finite State Automaton (FSA)



2.2 Mock del sistema

Dopo aver effettuato un'attenta analisi dei requisiti, con il supporto del committente, si é realizzata una prima architettura del sistema. In particolare questa risulta utile per modellare, rappresentare ed esplicare in modo formale i requisiti e il problema sopra delineato.

In particolare si é utilizzato un sistema un simulatore di un mondo virtuale fornito dalla nostra software house (Figura: 8.1); abbiamo creato un layer tra questo ed il nostro sistema in modo da creare una connessione per potervi comunicare.

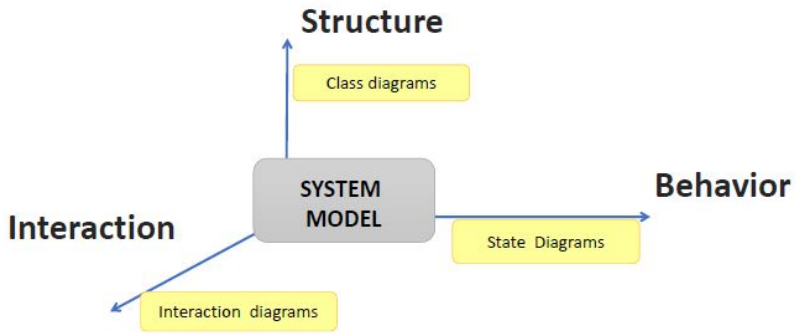


Figura 2.1: Definizione struttura, interazioni e comportamenti per mantenimento technology independent

2.3 Introduzione ai QActor

Per la rappresentazione *formale* del sistema si é deciso di utilizzare i **QActor**. In particolare, questo é un **meta-modello** ispirato al modello ad attori (supportato tramite le librerie akka), inoltre, il linguaggio qa é un linguaggio custom che ci permette di esprimere in modo conciso: struttura, interazione e comportamento dei sistemi software (in particolare distribuiti) i cui componenti interagiscono fra di loro usando messaggi ed eventi. Infine questo **meta-linguaggio** ci permette di superare le limitazioni della modellazione UML, che nasce principalmente object-oriented e con notevoli limitazioni nel distribuito. Tale meta-linguaggio ci é stato fornito dal nostro committente.

2.4 Una prima architettura logica

Durante le fasi di analisi (sia dei requisiti che del problema) si cerca di il più possibile **technology independent**, pur mantenendo **technology aware**. In particolare si usano i formalismi dei *qa* che ci permettono di definire: struttura, iterazioni e comportamenti [Figura 2.1] fra le varie entità che compongono il nostro sistema senza dover specificare in che modo verranno realizzate.

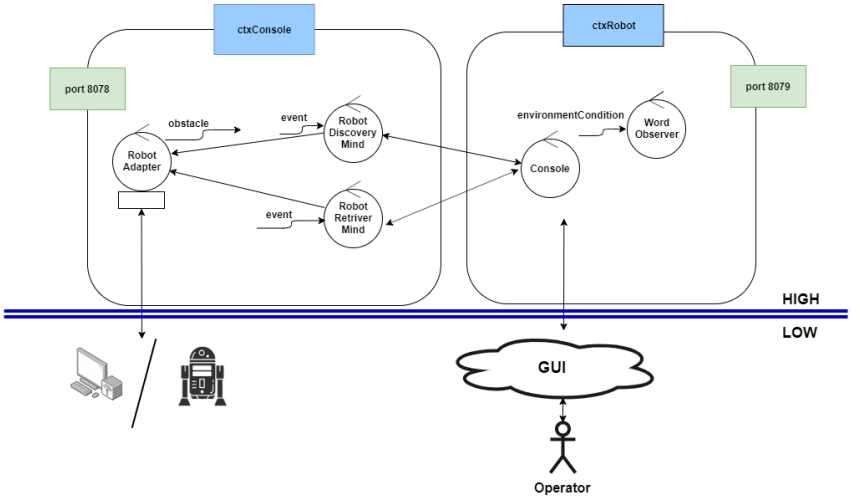


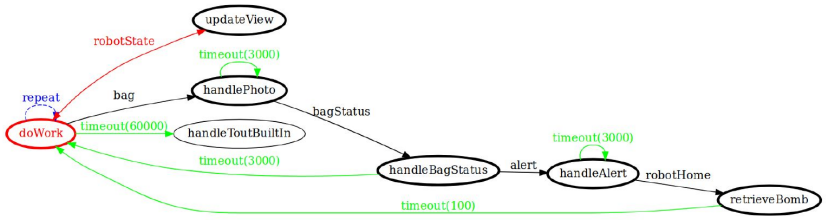
Figura 2.2: Architettura ddr system

2.4.1 Finite State Automaton (FSA)

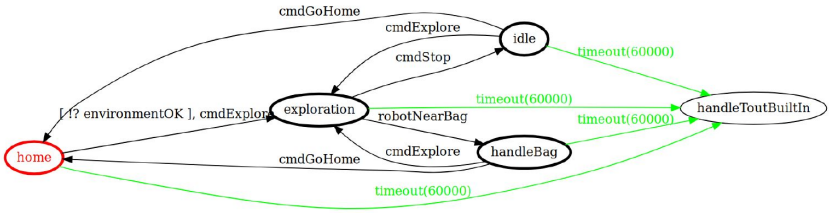
Usando approccio **top-down**, si sono definite le tre entità principali e si sono rappresentate usando i FSM [Figura 2.4.1, Figura 2.4.1, Figura 2.4.1].

In particolare Analizzando i tipi di interazioni si è giunti a determinare che robot-console hanno una comunicazione con relazione `1:1`, e quindi usiamo messaggi di tipo: **dispatch** per rendere la comunicazione diretta e sicura.

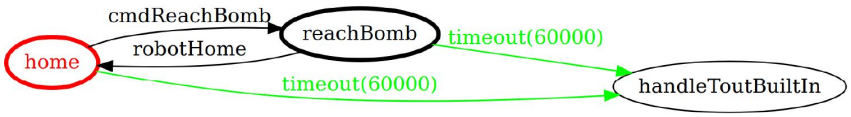
Console



Robot Discovery



Robot Retriever



2.5 Prima implementazione

```
1  System robot
2
3  /* Environment control  */
4  Dispatch environment: environment          // R-startExplore
5
6  /* Temperature control event */
7  Event temperature: temp(X)               // R-tempOk
8
9  /* Robot command */
10 Dispatch exploreCmd: exploreCmd            // R-explore
11 Dispatch stopCmd: stopCmd               // R-stopExplore
12 Dispatch backHomeCmd: backHomeCmd          // R-backHome
13 Dispatch continueExploreCmd: continueExploreCmd     // R-continueExplore
14
15 Dispatch bag: bag(X)
16
17 /* Console command */
```

```
Dispatch alert: alert               // R-alert
Dispatch bombInBag: bombInBag(X)

Dispatch robotState: robotState(X)
Dispatch robotBackHome: robotBackHome         // R-BackHome

Context ctxRobotImpl ip [ host='localhost' port=8079 ]

QActor robotretrieval context ctxRobotImpl {

  /* Rating initial condition (enviromentCondition)  */
  State home initial [

  ] transition
    stopAfter 10000
    whenMsg continueExploreCmd -> continueExplore                    // R-reachBag

  /* Business Logic */
  State continueExplore [
    // reach bomb and return at home        // R-reachBag & R-bagAtHome
  ] transition
    stopAfter 10000
    whenMsg robotBackHome -> home
}

QActor robotdiscovery context ctxRobotImpl {

  /* Rating initial condition (enviromentCondition)  */
  State home initial [

  ] transition
    stopAfter 10000
    whenMsg [ !? environment ]  exploreCmd -> exploration       // R-explore

  /* Business Logic */
  State exploration [
    // explore the environment
  ] transition
    stopAfter 10000
    whenMsg stopCmd -> idle           // R-stopExplore

    State idle [
    // halt and evaluate environment
  ] transition
    stopAfter 10000
    whenMsg exploreCmd -> exploration,        // R-continueExplore
    whenMsg backHomeCmd -> home         // R-backHome

  State handleBag [
    // halt near the bag and        // R-stopAtBag
    // take picture               // R-takePhoto
    forward console -m bombInBag: bag(X)  // R-sendPhoto
  ] transition
    stopAfter 10000
    whenMsg backHomeCmd -> home,
    whenMsg exploreCmd -> exploration
}

QActor console context ctxRobotImpl {

  State handleWork initial [

  ] transition
    stopAfter 10000
    whenMsg robotState -> updateView,     // R-consoleUpdate
    whenMsg bombInBag -> handlePhoto
    finally repeatPlan

  State handlePhoto [
    onMsg bag: bag(X) -> selfMsg bombInBag: bombInBag(false) // ACTION: evaluate bomb
```

```
  ] transition
    whenTime 3000 -> handlePhoto
    whenMsg bombInBag -> handleBagStatus

  State handleBagStatus [
    onMsg bombInBag: bombInBag(true) -> {
      // store a photo          // R-storePhoto
      forward robotdiscovery -m backHomeCmd: backHomeCmd;         // R-backHomeSinceBomb
      selfMsg alert: alert        // R-alert
    };
    onMsg bombInBag: isBomb(false) ->
      forward robotdiscovery -m exploreCmd: exploreCmd  // R-continueExploreAfterPhoto
  ] transition
    whenTime 3000 -> handleWork
    whenMsg alert -> handleAlert

  State handleAlert [

  ] transition
    whenTime 3000 -> handleAlert
    //whenMsg robotHome -> retrieveBomb     // R-whaitForHome

  State retrieveBomb [
    forward robotretrieval -m continueExploreCmd: continueExploreCmd  // R-reachBag
  ] transition
    whenTime 100 -> handleWork

  State updateView resumeLastPlan [
    onMsg robotState: state(X) -> printCurrentMessage
  ]
}
```

Capitolo 3

Sprint 3

Durante questo sprint prima ci si é preoccupati di collegare il robot fisico e iniziare a fare test del codice già presente.
Poi ci siamo dedicati alla creazione e l'inserimento nel sistema di un *planner* (requisito **R-explore**), che ci é stato in parte fornito dalla software house.

3.1 Planner

Si é voluto lavorare ad un planner che permettesse, in una fase iniziale, al nostro robot (fisico e virtuale) di riuscire ad esplorare, in completa autonomia una stanza vuota.
Tale planner poi é stato ampliato in modo da poter evitare gli ostacoli, analizzare le bombe e mantenere la memorizzazione per poter permettere al robot-retriver di recuperare la bomba conoscendo il giusto percorso.

3.1.1 Approccio: divisione dello spazio in celle

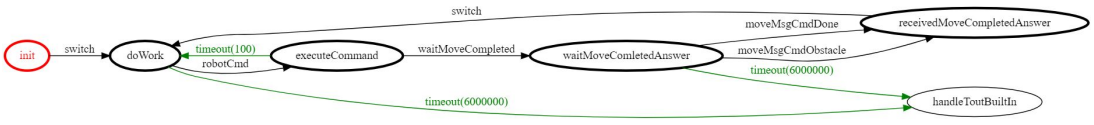
Ci siamo chiesti come risolvere il problema della gestione della navigazione del robot e di come renderlo "consapevole" della sua posizione nello spazio.
Per risolvere questo problema, si **suddividere lo spazio in celle**, ognuna della medesima dimensione del robot, al fine di permettergli la navigazione di cella in cella.
Tale approccio é dovuto all'utilizzo del planner aziendale, il quale ci impone una tecnica di gestione dello spazio ben specifica: la mappa é costituita da celle ed il planner fornisce l'indicazione dei movimenti da cella a cella.

3.1.2 Approccio: unità di misura, il tempo

Data che attualmente il committente non vuole dotare il robot di un GPS, o di altro utile per la nostra causa, l'unità di misura a nostra disposizione per la gestione dei movimenti atomici é il **tempo**.
Premettendo che, il robot conosce il tempo necessario per spostarsi da una cella all'altra, si deciso che:

- il robot si muove contando il tempo trascorso su una cella;
- successivamente il robot si sposta su un'altra cella.

Finite State Automa (FSA) - Planner



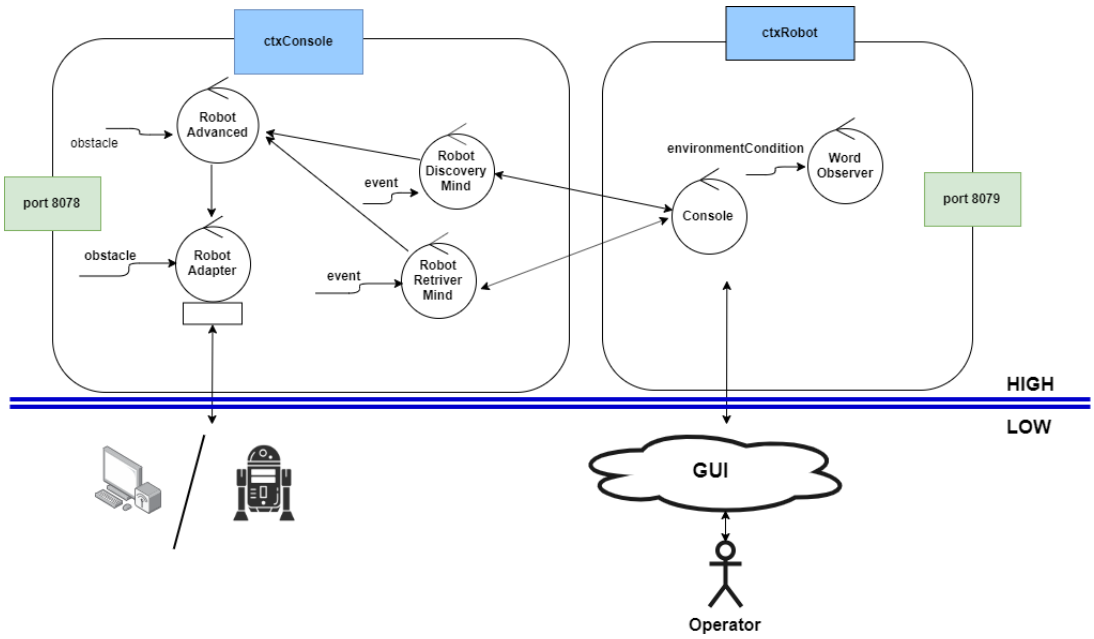
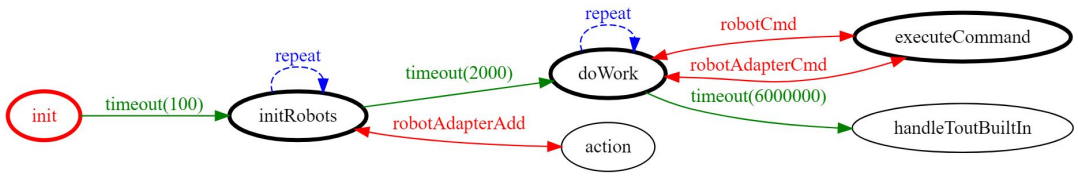


Figura 3.1: Architettura ddr system con robot-planner

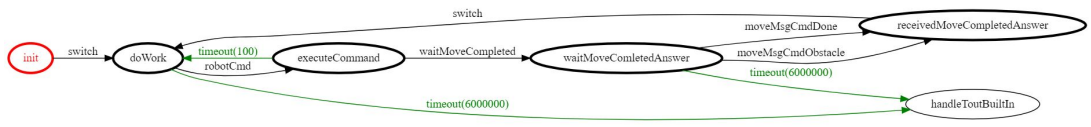
3.1.3 Approccio: robot-planner

Definito tale sistema di planning risulta quindi ovvio che é compito del robot sapere come muoversi da una cella all'altra in maniera atomica.
Da subito ci é stato chiaro che il nostro robot (robot-adapter), non fosse sufficiente ad eseguire tale compito. Infatti esso é capace di eseguire solo comandi semplici (avanti, indietro, destra, sinistra). Per tali motivazioni, si é deciso di aggiungere una nuova componente: **robot-advanced**, che sarà in grado di svolgere movimenti atomici sfruttando il robot adapter a livello inferiore (Figura 3.2.1).

Finite State Automa (FSA) - Robot Adapter



Finite State Automa (FSA) - Robot Advance



3.2 Prima implementazione

```
QActor robot_planner context ctxRobot {

  Rules {
    timew(255).
    timeTurn(500).
  }

  State init initial [

  ] switchTo doWork

  State doWork [

  ] transition
    stopAfter 60000
    whenMsg robotCmd -> executeCommand

  State executeCommand [
    onMsg robotCmd: robotCmd(a) -> {
      [ !? timeTurn(T) ] forward robot_adapter -m robotAdapterCmd: robotCmd(a,T);
      selfMsg waitMoveCompleted: waitMoveCompleted
    };
    onMsg robotCmd: robotCmd(d) -> {
      [ !? timeTurn(T) ] forward robot_adapter -m robotAdapterCmd: robotCmd(d,T);
      selfMsg waitMoveCompleted: waitMoveCompleted
    };
    onMsg robotCmd: robotCmd(w) -> {
      //time unit for movements
      [ !? timew(T) ] forward onecellforward -m moveMsgCmd : moveMsgCmd(T);
      selfMsg waitMoveCompleted: waitMoveCompleted
    }
  ] transition
    whenTime 100 -> doWork
    whenMsg waitMoveCompleted -> waitMoveComletedAnswer

  State waitMoveComletedAnswer [
  ] transition
    stopAfter 60000
    whenMsg moveMsgCmdDone -> receivedMoveCompletedAnswer,
    whenMsg moveMsgCmdObstacle -> receivedMoveCompletedAnswer

  State receivedMoveCompletedAnswer [
    onMsg moveMsgCmdDone: moveMsgCmdDone(X) ->
      replyToCaller -m moveMsgCmdDone: moveMsgCmdDone(X);
    onMsg moveMsgCmdObstacle: moveMsgCmdObstacle(moveWDuration(T)) ->
      forward robot_adapter -m robotAdapterCmd: robotCmd(s,T);
    onMsg moveMsgCmdObstacle: moveMsgCmdObstacle(moveWDuration(T)) ->
      replyToCaller -m moveMsgCmdObstacle: moveMsgCmdObstacle(moveWDuration(T))
  ] switchTo doWork
}
```

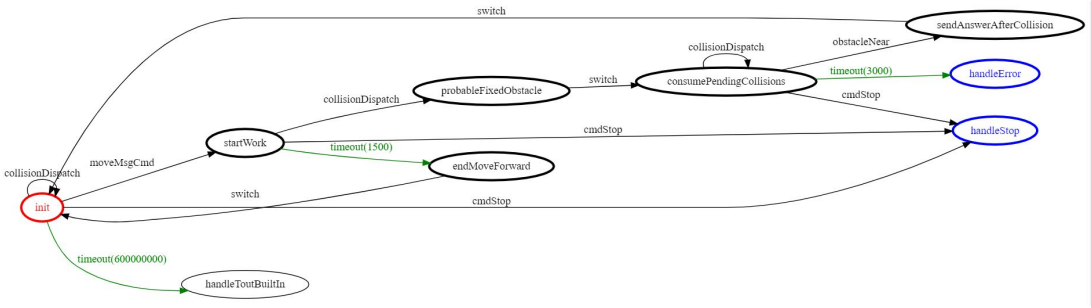
3.2.1 Approccio: one cell forward

Dato che l'esecuzione del comando per lo spostamento in *avanti* può causare lo scontro del robot con un ostacolo, deve essere possibile per il robot riposizionarsi nella cella dalla quale era partito, senza lasciarlo in stato inconsistente.

In particolare:

- **Robot virtuale**: non essendo dotato di sonar, non percepisce un ostacolo, quindi non lo rivelerà fintanto che non ci andrà a sbattere.
- **Robot fisico**: é dotato di sonar, dovrebbe essere capace di percepire l'ostacolo da prima di partire;

Finite State Automa (FSA)



Capitolo 4

Sprint 4

Durante questo sprint si é conclusa l'implementazione dell'interfaccia grafica.

4.1 Console

Come richiesto dal committente in console é possibile visualizzare:

- lo stato del robot;
- la posizione e movimento del robot;
- la temperatura;
- la mappa dinamica della Hall.

Inoltre deve essere possibile:

- settare la temperatura dell'ambiente;
- prendere il controllo del robot da remoto.

4.1.1 Considerazioni

Per rispettare il requisito **R-consoleUpdate** si deve prevedere una metodologia capace di inviare alla console la situazione attuale della hall, e quindi del mondo che circonda il robot. Inoltre, le entità che sono "gestiti" dalla console e che devono poter usare la mappa sono due: robot-discovery e robot-retriever, risulta logico sfruttare la conoscenza della console per memorizzare al suo interno una copia dello stato del sistema.

Altra peculiarità, risiede sul *tempo di aggiornamento* della console, per tale motivo si é deciso che il robot invierà la nuova conoscenza parziale dello stato alla console ad ogni modifica di questo. In altre parole, dato che il robot sa esattamente il tempo richiesto per spostarsi e il tempo di permanenza, mentre la console no, viene demandato a lui il compito di notificare il cambiamento di stato. In egual modo, ad ogni cambiamento di stato, la console notificherà i cambiamenti alla GUI (interfaccia grafica).

Infine ci é stato richiesto dal committente di abbiamo deciso di rendere visualizzabili i pulsanti presenti sulla console, solamente quando é possibile eseguire l'azione ad essi associata.

4.1.2 Tecnologie

Per la realizzazione della console si é deciso di usare:

- **Client-side**: Angular.
- **Server-side**: Node + Express, in quanto la software house possedeva già una parte di sistema.

4.2 Canali di comunicazione

Sostanzialmente si é deciso di implementare la comunicazione tramite: **WebSocket** e **MQTT**

4.2.1 WebSocket

Per permettere l'interazione dell'interfaccia grafica con il sistema si é deciso di usare le **WebSocket**, canali di comunicazione bidirezionali che utilizzano socket http.
Tale decisione é stata presa in quanto: il committente possedeva un server già implementato, inoltre, tale soluzione, ci permette di aggiornare *dinamicamente* la GUI ogni qualvolta lo stato del sistema cambia, senza dover implementare un meccanismo di *polling* da parte dell'utente.

4.2.2 MQTT: Message Queue Telemetry Transport

Per far comunicare l'*attore* incaricato di adattare la console con le GUI si é deciso di adottare il **protocollo MQTT** (Figura 4.1). Tale scelta é stata prese, in primo luogo, in un ottica **distribuita**, infatti così facendo risulta molto semplice l'aggiunta di più interfacce grafiche.

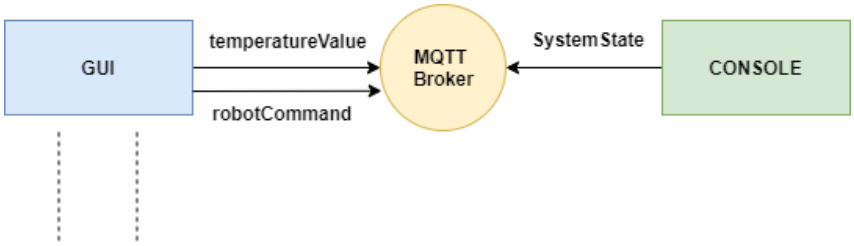


Figura 4.1: MQTT: comunicazione GUI e Console

Come mostrato in figura 4.1, la comunicazione é bidirezionale:

- **Console-GUI**: vengono inviate le informazioni sullo stato del sistema, utili per caricare i dati presenti sulla GUI.
- **GUI-Console**: per inviare i comandi.

4.3 Temperatura

Dato che il committente non ha espresso particolari requisiti a riguardo di come viene recuperata l'informazione a riguardo della temperatura, si é deciso di inviare tale informazione ad un evento su **MQTT**. A tal fine é stato realizzato un servizio capace di emettere questa informazione a comando dell'utente. Tale strumento è stato integrato nell'interfaccia grafica.

4.3.1 Modifica di requisito

Durante gli incontri con il committente, era sorto il problema di decidere se il robot-discovery dovesse tener contro della temperatura non solo nella fase di avvio del sistema, ma anche durante la fase di esplorazione.

Tale richiesta sarebbe facilmente integrabile all'interno del nostro sistema, andando a valutare tale variabile per ogni stato in uscita, invece che solo in fase di inizializzazione . Ma si é comunque deciso con il committente di ignorarla.

4.4 Interfaccia grafica

In figura 8.4 é possibile vedere l'interfaccia grafica.

Capitolo 5

Sprint 5

In questo sprint, ci siamo dapprima preoccupati di integrare nel nostro sistema anche il robot-retriever. A tale scopo ci si é preoccupati di: gestire il meccanismo di fotografie per cercare la bomba, gestire la mappa condivisa fra le due tipologie di robot e finire l'implementazione del recupero della bomba.

Inoltre si é migliorata l'interfaccia grafica, al fine di renderla più *user-friendly*.

5.1 Fotografia della borsa

Dato che da requisiti sappiamo che, il robot-discovery ogni qual volta incontra un ostacolo deve fermarsi (requisito **R-stopAtBag**), scattare una foto alla borsa (requisito **R-takePhoto**), inviare alla consolo tale immagine (requisito **R-sendPhoto**) al fine di renderla disponibile all'operatore.

In tale accezione da subito é sorto il problema di come trattare il perimetro della nostra hall, quindi di come trattare il muro. Per semplicità si é deciso di analizzare il *muro come borsa*, in particolare mi verrà notificato sempre che esso non contiene una bomba.

5.1.1 Problema: dotazione della fotocamera

Al momento il nostro robot non risulta dotato di fotocamera. A tal fine si é deciso con il committente di **simulare** la fotocamera, caricando delle immagini sul robot-discovery, al fine di testare il giusto comportamento del sistema.

Per tali motivi si é deciso di codificare le immagini in **base64** dal robot, per poi inviarla alla console insieme alle informazioni dello stato (in formato testuale).

5.2 Informazioni relative allo stato

Oltre alle informazioni base dello stato del sistema, ci é stato richiesto dal nostro committente l'invio di altre informazioni aggiuntive:

- stato del robot,
- alert nel caso in cui temperatura aumenti;
- immagine bomba.

Dato che il nostro committente non specificato nessuna limitazione, si é deciso di prendere la strada più semplice ed inviare queste informazioni insieme alle informazioni riguardanti lo stato del sistema. A tale scopo quindi, sarà la console che si preoccuperà di inviare periodicamente alla GUI queste informazioni, a seguito di un cambiamento di stato.

5.3 Storage delle foto

Dal requisito **R-storePhoto** sappiamo che, a seguito di un invio di una foto da parte del robot-discovery, é necessario memorizzare la foto ricevuta al fine di farla analizzare dall'algoritmo messo a disposizione dalla nostra software house.
A tal fine, per limitare gli scambi in rete e non intasare il nostro sistema di storage, si é deciso di memorizzare **temporaneamente** sulla console le foto ricevute dal sistema, e solo in caso questa presenti una bomba sarà memorizzata **permanentemente** su uno storage apposito.

Per quanto riguarda il problema della memorizzazione "fisica" delle foto, risulta possibile eseguirla:

- sul robot;
- sulla console;
- su un servizio esterno.

Analizzando le varie opportunità si é da subito esclusa la soluzione riguardante il robot, esso infatti conosce solo la foto, e siccome da requisiti é stato richiesto anche la memorizzazione delle **informazioni contestuali**, ci é sembrato non consono. Per quanto riguarda invece la memorizzazione in remoto (console o servizio esterno) non é stato specificato nessun interesse da parte del committente, quindi si é deciso di adottare la soluzione più semplice, memorizzare le informazioni sulla *console* per poi inviarle in formato testuale insieme alle informazioni riguardanti lo stato del sistema.

5.4 Robot retriever

Come si può già notare dalla figura 7.1 il **robot-retriever** é stato realizzato utilizzando gli stessi strumenti utili per creare il robot-discovery.
In questo contesto é però necessario sottolineare che il robot-retriever non deve esplorare la hall, infatti esso, come da requisito (**R-reachBag**) non deve incontrare ostacoli e deve già conoscere la strada per andare a recuperare la borsa contenente la bomba e riportarla a casa (requisito **R-bagAtHomes**).
A tale scopo, dato che la console possiede una copia dello stato del sistema sarà direttamente la console che deve dotare il robot-retriever della conoscenza del mondo prima del suo avvio (**R-reachBag**), senza ulteriori scambi intermedi.

Altra considerazione, molto importante é che, si lavora con la premessa che il mondo sia **statico**. Anche se quindi, teoricamente nulla nell'ambiante dovrebbe cambiare, sarebbe buona norma e cosa prevedere una gestione in caso di errori in fasi di planning, che ci tiene quindi anche già aperti ad una situazione in qui gli ostacoli siano dinamici.

5.4.1 Recupero della borsa: sviluppo futuro

Questione braccio meccanico

5.5 Gestione della mappa

Avendo ampliato il nostro sistema inserendo il *Robot retriever* un problema cruciale su cui discutere e come gestire la condivisione della mappa.

A tale scopo, si é pensato aggiungere tale mappa alla sua conoscenza di base nel momento dell'avvio in modo tale da permettergli di recuperare la borsa contenente la bomba.

5.5.1 Condivisione della mappa

Per permettere la condivisione della mappa con il robot-retriever sono possibili sostanzialmente due soluzioni:

- il robot-discovery comunica al robot-retriever le informazioni necessarie;
- la console comunica al robot-retriever le informazioni necessarie.

Per una maggiore elasticità, si é deciso di implementare la seconda soluzione, ovvero sarà la **console** a inviare le informazioni sulla mappa. Infatti così facendo: la console rimane un punto centrale di controllo e l'avvio del robot-retriever sarà più tempestivo, in quanto dopo che il robot-discovery giunge a casa dopo aver intercettato la bomba (requisito **R-backHomeSinceBomb**) il robot-retriever possa subito partire in quanto possiede già la conoscenza necessaria per raggiungere la bomba.

5.5.2 Tipo di comunicazione

Si è deciso da subito di condividere la mappa tramite **messaggi**. Tale scelta é stata dettata da una scelta orientata alla *sicurezza*, infatti si vuole che solo il robot-retriever venga a conoscenza di tale informazione.

5.6 Un implementazione

```
QActor robot_retriever_mind context ctxRobot {

  Rules {
    environment(notok).

    eval(eq, X, X).   //since we have syntax limitations
    doTheMove(M) :-
      move(M1), !,
      eval(eq,M,M1), !,
      doTheFirstMove(M).

    doTheFirstMove(w) :- retract( move(w) ),!.
    doTheFirstMove(a) :- retract( move(a) ),!.
    doTheFirstMove(d) :- retract( move(d) ),!.

    homeReady :- curPos(0,0,D), bomb(_,_).
    bombInRoom :- bomb(_,_).

    nearBomb :- curPos(X,Y,_), eval(plus,X,1,R), bomb(R,Y).
    nearBomb :- curPos(X,Y,_), eval(minus,X,1,R), bomb(R,Y).
    nearBomb :- curPos(X,Y,_), eval(plus,Y,1,R), bomb(X,R).
    nearBomb :- curPos(X,Y,_), eval(minus,Y,1,R), bomb(X,R).
  }

  // Set planner in retriver knowledge
  State init initial [
    javaRun it.unibo.planning.planUtil.initAI()
  ] switchTo home

  State home [

  ] transition
    stopAfter 6000000
    whenEvent environment: environment(E) do
      demo replaceRule(environment(X), environment(E)),
    whenMsg map: map(M) do javaRun it.unibo.utils.updateStateOnRobot.loadMap(M),
    whenMsg cmdReachBomb -> checkTemperatureAndRetrieve   // R-reachBag & R-TempOk
    finally repeatPlan

  State checkTemperatureAndRetrieve [           // R-TempOk
    [ !? environment(ok) ] selfMsg cmdReachBomb: cmdReachBomb
  ] transition
    whenTime 100 -> home
```

```
    whenMsg cmdReachBomb -> goToReachBomb

  /*
   * Retrieval business logic
   */
  State goToReachBomb [
    println("STATE[goToReachBomb] ...");
      javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-retrieving"
          );
      [ !? bomb(X,Y)  ] javaRun it.unibo.planning.planUtil.setGoal(X,Y);
      javaRun it.unibo.planning.planUtil.doPlan()
  ] transition
    whenTime 1000 -> doActions

  State goToIdle [
    println("RETRIEVER_MIND[goToIdle] ...");
      [ !? nearBomb ] {
        javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-
            retrieving");
        delay 3000;
        removeRule bomb(_,_);
        selfMsg endAction: endAction
      } else {
        [ !? bombInRoom ] {
          javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-idle");
        forward robot_adapter -m robotCmd: robotCmd(blinkStop)          // R-blinkLed (
            stop)
        } else {
          javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-idle-
              with-bomb");
        forward robot_adapter -m robotCmd: robotCmd(blinkStop)          // R-blinkLed (
            stop)
        }
      }
  ] transition
    whenTime 100 -> idle
    whenMsg endAction -> goToHome

  State goToHome [
    println("STATE[goToHome] ...");
      javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-returning")
          ;
    forward robot_adapter -m robotCmd: robotCmd(blinkStart)           // R-blinkLed (
        start)
  ] switchTo backToHome

  State idle [
    // reach bomb solo se non ha bomba, go home in entrambi
    [ !? bombInRoom ] selfMsg idleWhileRetrieving: idleWhileRetrieving
      else selfMsg idleWhileReturning: idleWhileReturning
  ] transition
    stopAfter 6000000
    whenMsg cmdStop -> idle,
    whenMsg idleWhileRetrieving -> idleWhileRetrieving,
    whenMsg idleWhileReturning -> idleWhileReturning

  State idleWhileRetrieving [
      javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-idle-
          retrieving")
  ] transition
    stopAfter 6000000
    whenMsg cmdStop -> idleWhileRetrieving,
    whenMsg cmdReachBomb -> goToReachBomb,
    whenMsg cmdGoHome -> goToHome                        // R-backHome

  State idleWhileReturning [
      javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-idle-
          returning")
  ] transition
    stopAfter 6000000
```

```
    whenMsg cmdStop -> idleWhileReturning,
    whenMsg cmdGoHome -> goToHome                        // R-backHome

  /*
   * Planner
   */
  Plan doActions[
    [ !? move(M) ] println( doActions_doingTheMove(M) );
    [ not !? move(M) ] selfMsg endAction : endAction ;
      [ !? move(M) ] selfMsg waitMoveCompleted: waitMoveCompleted;
      [ !? doTheMove(M) ] forward robot_planner -m robotCmd: robotCmd(M)
  ] transition
    stopAfter 6000000
    whenMsg cmdStop -> goToIdle,
    whenMsg waitMoveCompleted -> waitMoveCompletedAnswer,
      whenMsg endAction -> backToHome
      finally repeatPlan

  Plan waitMoveCompletedAnswer [

  ] transition
    stopAfter 6000000
    whenMsg moveMsgCmdObstacle -> goToIdle,
    whenMsg cmdStop -> goToIdle,
    whenMsg moveMsgCmdDone -> handleCmdDone

  Plan handleCmdDone [
    printCurrentMessage;
    onMsg moveMsgCmdDone: moveMsgCmdDone(X) ->
      javaRun it.unibo.planning.planUtil.doMove(X); //update the map
    javaRun it.unibo.planning.planUtil.showMap();
      javaRun it.unibo.utils.updateStateOnConsole.updateMap()
  ] transition
    whenTime 100 -> doActions
    whenMsg cmdStop -> goToIdle

  Plan backToHome  [
    [ !? curPos(0,0,D)  ]{
      forward robot_adapter -m robotCmd: robotCmd(blinkStop);  // R-blinkLed (stop)
      println("AT HOME");
      javaRun it.unibo.planning.planUtil.showMap();
        javaRun it.unibo.utils.updateStateOnConsole.updateMap();
      [ !? homeReady ] selfMsg cmdGoHome: cmdGoHome
        else selfMsg endAction: endAction
     }
    else{
        javaRun it.unibo.planning.planUtil.setGoal("0","0");
        [ !? curPos(X,Y,D)  ]  println( backToHome(X,Y,D) );
        javaRun it.unibo.planning.planUtil.doPlan()
      }
  ] transition
    whenTime 100 -> doActions
    whenMsg endAction -> terminate,
    whenMsg cmdGoHome -> home,
    whenMsg robotHomeAfterBomb -> home

  State terminate [
    println("STATE[terminate] retriever home!");
      javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("terminating")
  ]

  Plan handleError[ println("mind ERROR") ]

}
```

Capitolo 6

Sprint 6

6.1 Sistema distribuito

In questa fase della realizzazione del sistema siamo andati a delocalizzare la nostra soluzione separando il sistema in due nodi principali: il **robot** e la **console**.

In particolare:

- Robot: contiene tutta la gestione del robot (adapter, advance, planner, oncellforward). Ma in esso troviamo anche la **business logic** del suo funzionamento, che é rappresentata quindi dalla *mind*.
- Console: contiene la gestione della console e il word-observer, che ci permette di valutare ed elaborare tutti gli stimoli esterni (condizioni ambientali).

Capitolo 7

Il sistema

In questa sezione si andrà a definire la struttura del sistema finito.

7.1 Architettura logica

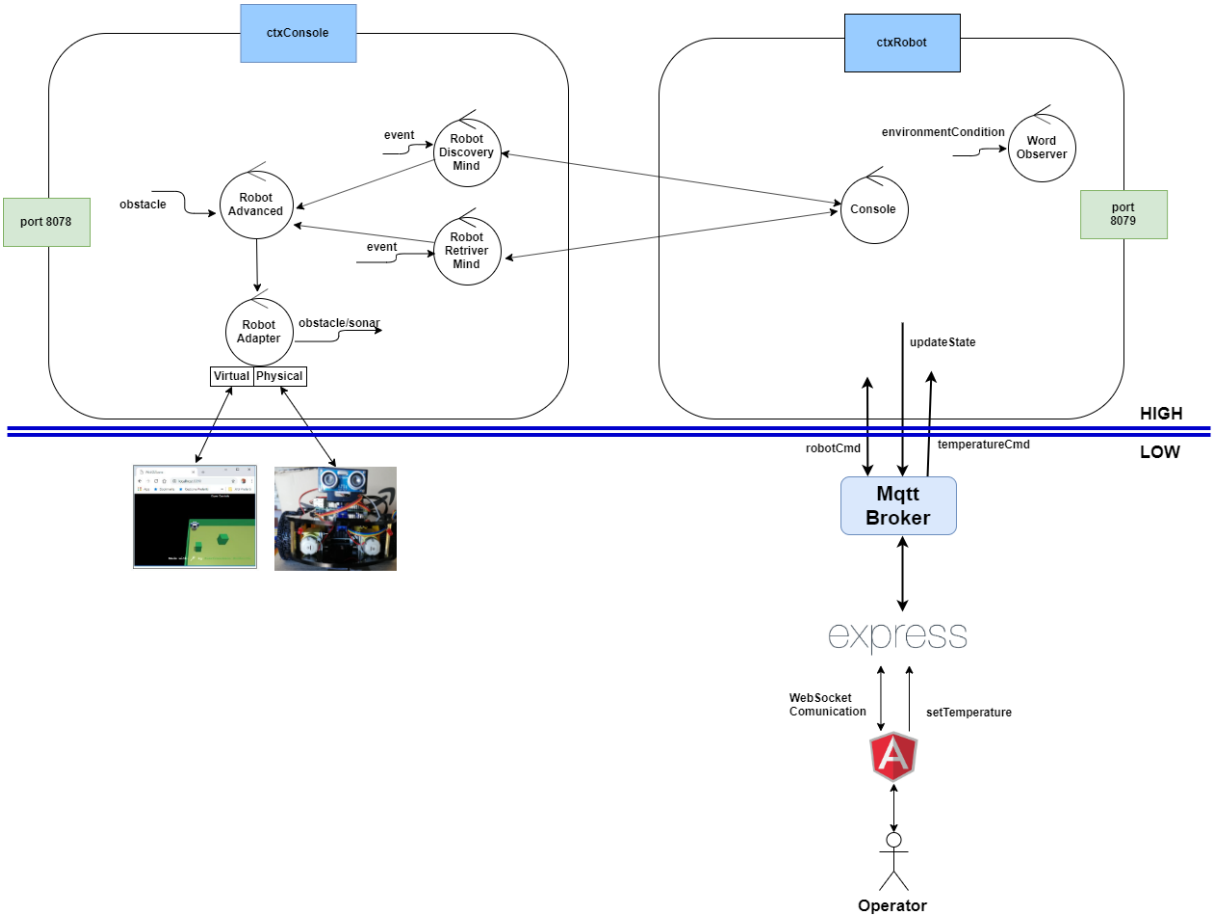
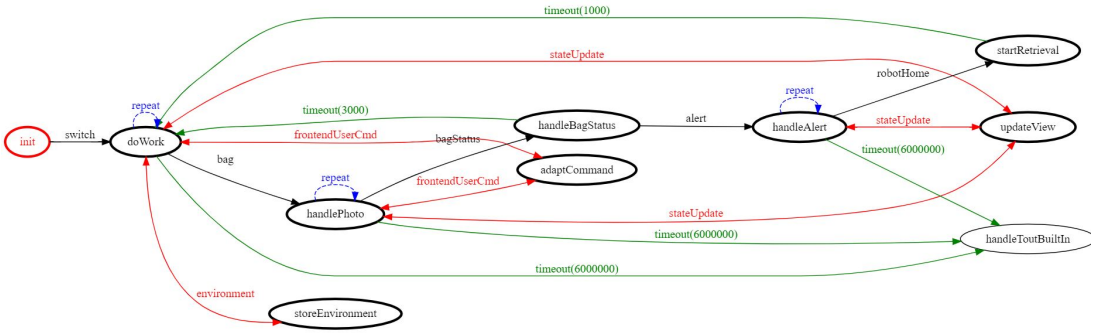


Figura 7.1: Architettura ddr system

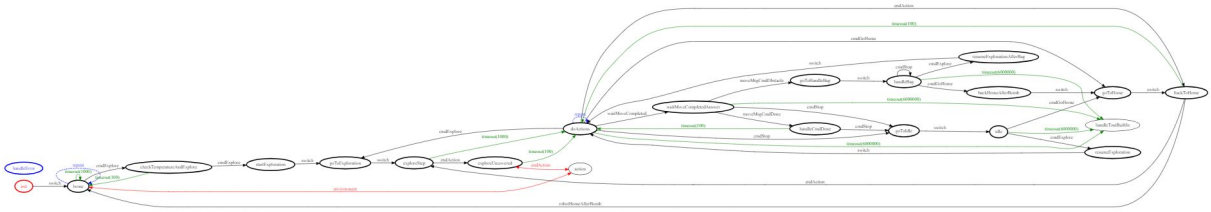
7.1.1 Finite State Automaton (FSM)

Di seguito vengono riportati i diagrammi relativi alle 3 componenti principali finite.

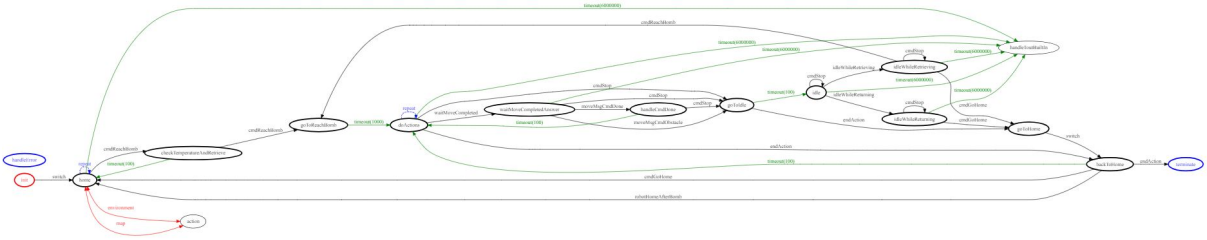
Console



Robot Discovery Mind



Robot Retriever Mind



7.1.2 Interazioni

Quindi in conclusione:

- La comunicazione dei **robot-mind** con il **robot-adapter** avviene tramite *scambio di messaggi*. Tale decisione é dettata dal fatto che non si vogliono "perdere" informazioni utili e/o urgenti.
- La comunicazione fra i due contesti principali, **robot** e **console** avviene tramite scambio di messaggi.
- Il **world-observer** ha il compito di "osservare" le condizioni ambientali. Quindi in questo caso risulta necessario *emettere e ricevere eventi*, sia da parte del word-observer, sia da parte del sistema (per conoscere le condizioni).
- La comunicazione fra **console** e **front-end** avviene tramite protocollo *MQTT*.
- Il front-end é composto da **Angular** e da **Express**. Questi 2 framework comunicano tramite servizi e *web socket*.

Capitolo 8

Tecnologie

In questo capitolo vengono elencate tutte le tecnologie utilizzate per l'implementazione della nostra soluzione.

8.1 QActor

Per realizzare il sistema abbiamo usato la stessa software factory usata per la rappresentazione formale del sistema: i **QActor**. In questo contesto, i QA vengono usati come linguaggio per la *generazione automatica* di codice per sistemi distribuiti. Grazie a tale soluzione é stato possibile ridurre nettamente i tempi di sviluppo del intero sistema.

8.2 Robot Virtuale

Simulazione di un robot [Figura 8.1] creato tramite il runtime JavaScript: Node.js. Tale soluzione ci é stata fornita dalla nostra software house.
Tale soluzione di ha permesso di sviluppare il nostro sistema in totale indipendenza dai vincoli fisici, infatti l'integrazione con il robot fisico é avvenuta solo durante l'ultimo sprint.

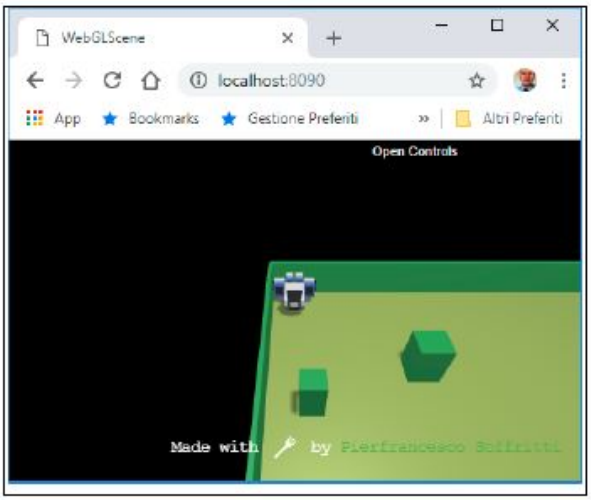


Figura 8.1: Soffritti: un simulatore virtuale

8.3 Robot Fisico

Questo robot [Figura 8.3] é stato direttamente assemblato durante o sviluppo. Per una maggiore semplicità e usabilità si é deciso di realizzarlo con **Arduino**.
Esso é dotato di:

- 4 motori e 4 ruote per permettere la navigazione.
- 1 sonar frontale per individuare gli ostacoli.
- 2 led per una comunicazione visuale con il mondo esterno.
- 1 bluetooth che ci permette di instaurare una connessione con esso e ci permette inoltre di avere una maggior libertà di movimento da parte del robot.

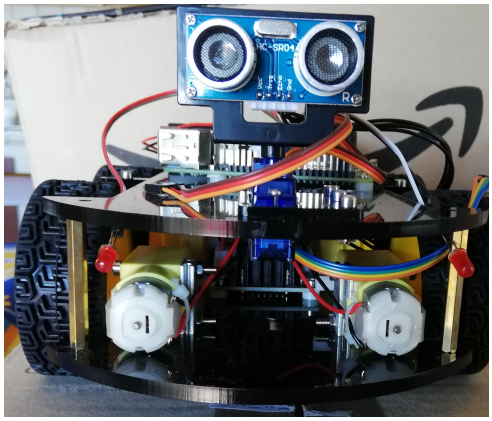


Figura 8.2: Robot fisico

8.3.1 Arduino: configurazione

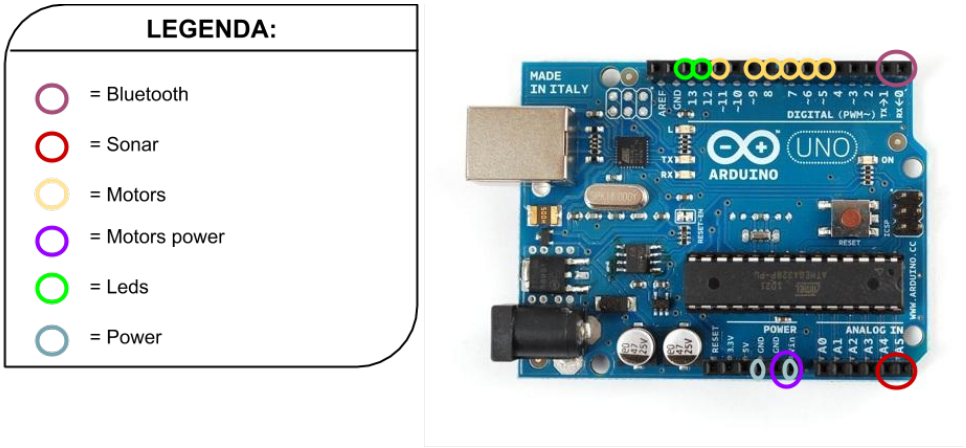


Figura 8.3: Configurazione pin arduino

8.4 MQTT

Da subito ci siamo posti la possibilità di poter gestire più di una interfaccia grafica. Per tale motivo si é deciso di usare il pattern **Publish-Subscribe**, in particolar modo nell'accezione di **Broker** MQTT, in quanto già fornito dalla nostra software factory.

8.5 Front-end

In figura 8.4 é presente uno screenshot dell'interfaccia grafica da noi implementata.

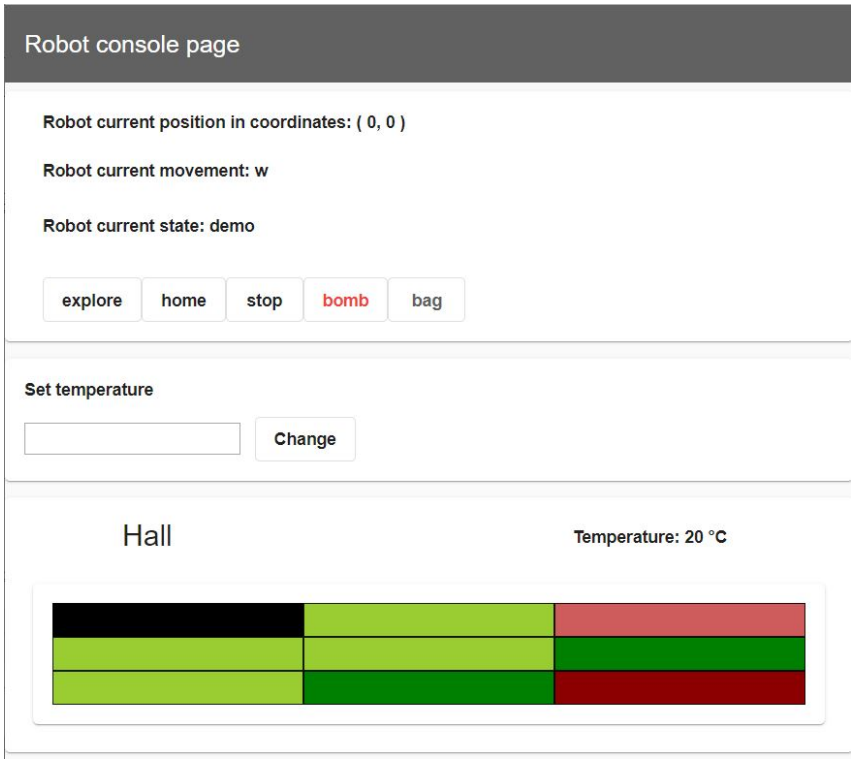


Figura 8.4: Interfaccia utente, guidata dall'operatore umano

8.5.1 Node

Partendo dall'interfaccia grafica fornita dai QA, abbiamo implementato una GUI (secondo il pattern MVC) adottando tecnologie che ci permettessero di gestire il pattern e l'aggiornamento automatico dell'interfaccia. Queste tecnologie sono presenti all'interno di Node.js e sono: **Express** e **Angular**.

8.5.2 Express

Il framework **Express** é stato utilizzato per sviluppare il server. Nello specifico questo si occupa di connettersi con il *broker MQTT* per comunicare con il sistema e per aprire una *Web-Socket* con il client, in modo da avere una comunicazione bidirezionale (senza dover fare polling).

8.5.3 Angular

Angular é stato usato per sviluppare l'interfaccia lato client. Questo ci ha ha permesso di ottimizzare ed automatizzare l'aggiornamento della GUI. In particolare 'e stato usato lo store per memorizzare il modello, e i vari componenti dell'interfaccia si sottoscrivono per essere aggiornati ad ogni modifica.

Capitolo 9

Note

9.1 MQTT

Durante lo sviluppo si é notato che il software messo a disposizione dalla nostra software factory presentava diversi problemi:

- Impossibilità di comunicazione tramite scambio di messaggi su Akka quando gli attori appartengono a contesti differenti.
- Perdita di pacchetti.

9.2 Attori dinamici

QA non permette la creazione dinamica di attori. Questo però sarebbe stato molto utile, ad esempio, per creare un robot adapter a run time, per ogni robot che richiedesse di

9.3 Problemi con Windows

I continui aggiornamenti (FORZATI) di Windows mi hanno causato un continuo riadattamento del sistema e causato parecchio ritardo. In particolar modo, ad ogni aggiornamento mi veniva “bloccata” la comunicazione fra raspberry e computer.