

## ***Robot Discovery - ddr***

*Elaborato di progetto per il corso Sviluppo di Sistemi Software*

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

Chiara Volonnino<sup>1</sup>

Email ids: <sup>1</sup>chiara.volonnino@studio.unibo.it

A.A. 2018/2019

# Indice

<b>1</b>	<b>Sprint 1</b>	<b>3</b>
1.1	Requisiti . . . . .	3
1.2	Analisi dei requisiti . . . . .	4
1.3	Analisi del problema . . . . .	5
<b>2</b>	<b>Sprint 2</b>	<b>7</b>
2.1	Mock del sistema . . . . .	7
2.2	Introduzione ai QActor . . . . .	7
2.3	Una prima architettura logica . . . . .	7
2.3.1	Finite State Automaton (FSM) . . . . .	9
2.4	Prima implementazione . . . . .	9
<b>3</b>	<b>Sprint 3</b>	<b>11</b>
3.1	Planner . . . . .	11
3.1.1	Approccio: divisione dello spazio in celle . . . . .	11
3.1.2	Approccio: unità di misura, il tempo . . . . .	11
3.1.3	Approccio: robot-planner . . . . .	12
3.2	Prima implementazione . . . . .	12
3.2.1	Approccio: one cell forward . . . . .	13
<b>4</b>	<b>Sprint 4</b>	<b>14</b>
4.1	Console . . . . .	14
4.1.1	Considerazioni . . . . .	14
4.1.2	Tecnologie . . . . .	14
4.2	Canali di comunicazione . . . . .	15
4.2.1	WebSocket . . . . .	15
4.2.2	MQTT: Message Queue Telemetry Transport . . . . .	15
4.3	Temperatura . . . . .	15
4.3.1	Modifica di requisito . . . . .	15
4.4	Interfaccia grafica . . . . .	15
<b>5</b>	<b>Sprint 5</b>	<b>16</b>
5.1	Fotografia della borsa . . . . .	16
5.1.1	Problema: dotazione della fotocamera . . . . .	16
5.2	Informazioni relative allo stato . . . . .	16
5.3	Storage delle foto . . . . .	17
5.4	Robot retriever . . . . .	17
5.5	Gestione della mappa . . . . .	17
5.5.1	Condivisione della mappa . . . . .	17
5.5.2	Tipo di comunicazione . . . . .	18
5.6	Un implementazione . . . . .	18
<b>6</b>	<b>Sprint 6</b>	<b>21</b>
6.1	Sistema distribuito . . . . .	21

<b>7</b>	<b>Il sistema</b>	<b>22</b>
7.1	Architettura logica . . . . .	22
7.1.1	Finite State Automaton (FSM) . . . . .	23
<b>8</b>	<b>Tecnologie</b>	<b>24</b>
8.1	QActor . . . . .	24
8.2	Robot Virtuale . . . . .	24
8.2.1	Soffritti . . . . .	24
8.3	Robot Fisico . . . . .	24
8.3.1	Arduino . . . . .	24
8.4	Console . . . . .	24
8.4.1	Node . . . . .	24
8.4.2	Express . . . . .	24
8.4.3	Angular . . . . .	24
<b>9</b>	<b>Note</b>	<b>26</b>

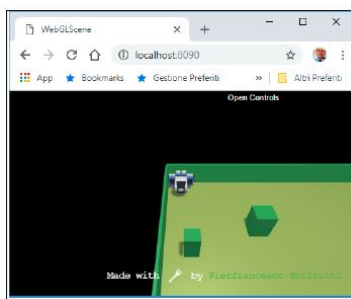
# Capitolo 1

## 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 riscontrata 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 riceve 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 su 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 **environnementCondition**. 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 avrà 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 è 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 termini 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**, 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**).

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 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 esprimere 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: 8.2); abbiamo creato un layer tra questo ed il nostro sistema in modo da creare una connessione per potervi comunicare.

### 2.2 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.

### 2.3 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 fra le varie entità che compongono il nostro sistema senza dover specificare in che modo verranno realizzate.

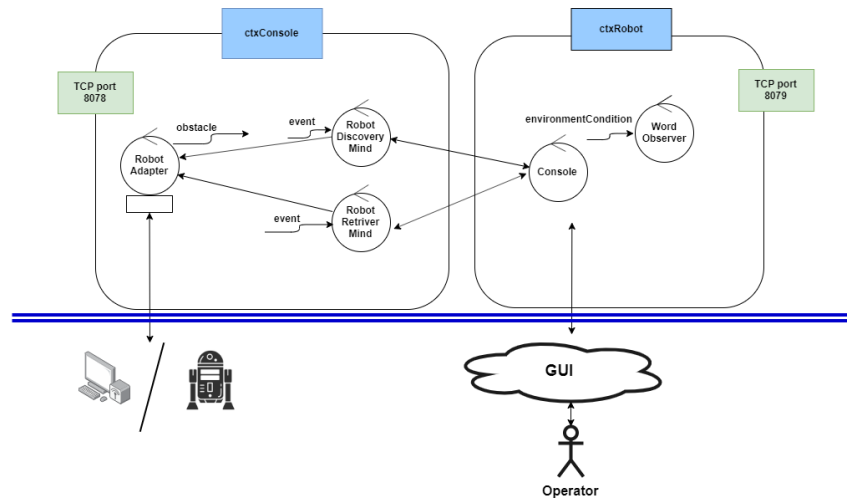


Figura 2.1: Architettura ddr system

### 2.3.1 Finite State Automaton (FSM)

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

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

## 2.4 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 */
18 Dispatch alert: alert // R-alert
19 Dispatch bombInBag: bombInBag(X)
20
21 Dispatch robotState: robotState(X)
22 Dispatch robotBackHome: robotBackHome // R-BackHome
23
24 Context ctxRobotImpl ip [ host='localhost' port=8079 ]
25
26 QActor robotretrieval context ctxRobotImpl {
27
28 /* Rating initial condition (enviromentCondition) */
29 State home initial [
30
31 ] transition
32 stopAfter 10000
33 whenMsg continueExploreCmd -> continueExplore // R-reachBag
34
35 /* Business Logic */
36 State continueExplore [
37 // reach bomb and return at home // R-reachBag & R-bagAtHome
38 ] transition
39 stopAfter 10000
40 whenMsg robotBackHome -> home
41 }
42
43 QActor robotdiscovery context ctxRobotImpl {
44
45 /* Rating initial condition (enviromentCondition) */
46 State home initial [
47
48 ] transition
49 stopAfter 10000
50 whenMsg [ !? environment ] exploreCmd -> exploration // R-explore
51 }
```

```

52  /* Business Logic */
53  State exploration [
54    // explore the environment
55  ] transition
56    stopAfter 10000
57    whenMsg stopCmd -> idle          // R-stopExplore
58
59    State idle [
60      // halt and evaluate environment
61    ] transition
62      stopAfter 10000
63      whenMsg exploreCmd -> exploration,      // R-continueExplore
64      whenMsg backHomeCmd -> home             // R-backHome
65
66  State handleBag [
67    // halt near the bag and          // R-stopAtBag
68    // take picture                  // R-takePhoto
69    forward console -m bombInBag: bag(X) // R-sendPhoto
70  ] transition
71    stopAfter 10000
72    whenMsg backHomeCmd -> home,
73    whenMsg exploreCmd -> exploration
74  }
75
76  QActor console context ctxRobotImpl {
77
78    State handleWork initial [
79
80    ] transition
81      stopAfter 10000
82      whenMsg robotState -> updateView,      // R-consoleUpdate
83      whenMsg bombInBag -> handlePhoto
84      finally repeatPlan
85
86    State handlePhoto [
87      onMsg bag: bag(X) -> selfMsg bombInBag: bombInBag(false) // ACTION: evaluate bomb
88    ] transition
89      whenTime 3000 -> handlePhoto
90      whenMsg bombInBag -> handleBagStatus
91
92    State handleBagStatus [
93      onMsg bombInBag: bombInBag(true) -> {
94        // store a photo          // R-storePhoto
95        forward robotdiscovery -m backHomeCmd: backHomeCmd;          // R-backHomeSinceBomb
96        selfMsg alert: alert      // R-alert
97      };
98      onMsg bombInBag: isBomb(false) ->
99        forward robotdiscovery -m exploreCmd: exploreCmd // R-continueExploreAfterPhoto
100    ] transition
101      whenTime 3000 -> handleWork
102      whenMsg alert -> handleAlert
103
104    State handleAlert [
105
106    ] transition
107      whenTime 3000 -> handleAlert
108      //whenMsg robotHome -> retrieveBomb      // R-whaitForHome
109
110    State retrieveBomb [
111      forward robotretrieval -m continueExploreCmd: continueExploreCmd // R-reachBag
112    ] transition
113      whenTime 100 -> handleWork
114
115    State updateView resumeLastPlan [
116      onMsg robotState: state(X) -> printCurrentMessage
117    ]
118  }

```

# Capitolo 3

## Sprint 3

Durante questo sprint prima ci si è 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 è stato in parte fornito dalla software house.

### 3.1 Planner

Si è 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 è 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 è dovuto all’utilizzo del planner aziendale, il quale ci impone una tecnica di gestione dello spazio ben specifica: la mappa è 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 è 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.

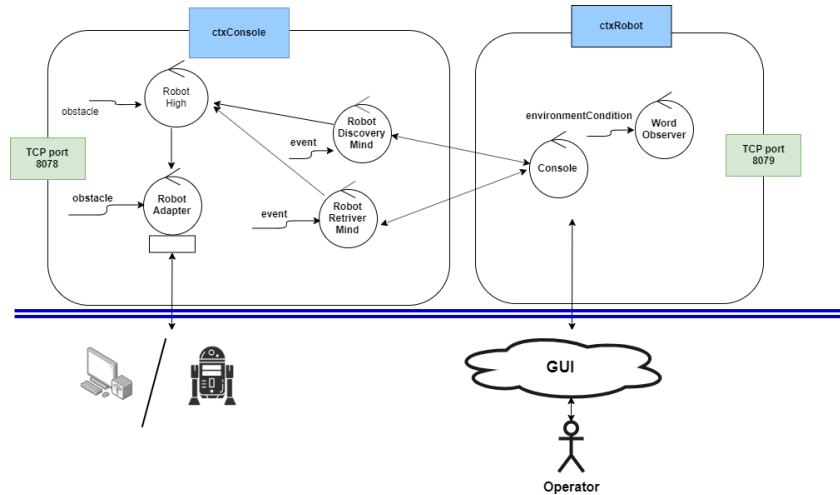


Figura 3.1: Architettura ddr system con robot-planner

### 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-planner**, che sarà in grado di svolgere movimenti atomici sfruttando il robot adapter a livello inferiore (Figura 3.1).

## 3.2 Prima implementazione

```

1  QActor robot_planner context ctxRobot {
2
3      Rules {
4          timew(255).
5          timeTurn(500).
6      }
7
8      State init initial [
9
10     ] switchTo doWork
11
12     State doWork [
13
14     ] transition
15     stopAfter 60000
16     whenMsg robotCmd -> executeCommand
17
18     State executeCommand [
19         onMsg robotCmd: robotCmd(a) -> {
20             [ !? timeTurn(T) ] forward robot_adapter -m robotAdapterCmd: robotCmd(a,T);
21             selfMsg waitMoveCompleted: waitMoveCompleted
22         };
23         onMsg robotCmd: robotCmd(d) -> {
24             [ !? timeTurn(T) ] forward robot_adapter -m robotAdapterCmd: robotCmd(d,T);
25             selfMsg waitMoveCompleted: waitMoveCompleted
26         };
27         onMsg robotCmd: robotCmd(w) -> {
28             //time unit for movements
29             [ !? timew(T) ] forward onecellforward -m moveMsgCmd : moveMsgCmd(T);
30             selfMsg waitMoveCompleted: waitMoveCompleted
31         }
32     ]
33 }

```

```

32 ] transition
33   whenTime 100 -> doWork
34   whenMsg waitMoveCompleted -> waitMoveCompletedAnswer
35
36 State waitMoveCompletedAnswer [
37   ] transition
38   stopAfter 60000
39   whenMsg moveMsgCmdDone -> receivedMoveCompletedAnswer,
40   whenMsg moveMsgCmdObstacle -> receivedMoveCompletedAnswer
41
42 State receivedMoveCompletedAnswer [
43   onMsg moveMsgCmdDone: moveMsgCmdDone(X) ->
44     replyToCaller -m moveMsgCmdDone: moveMsgCmdDone(X);
45   onMsg moveMsgCmdObstacle: moveMsgCmdObstacle(moveWDuration(T)) ->
46     forward robot_adapter -m robotAdapterCmd: robotCmd(s,T);
47   onMsg moveMsgCmdObstacle: moveMsgCmdObstacle(moveWDuration(T)) ->
48     replyToCaller -m moveMsgCmdObstacle: moveMsgCmdObstacle(moveWDuration(T))
49 ] switchTo doWork
50 }

```

### 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;

# 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 presa, in primo luogo, in un'ottica **distribuita**, infatti così facendo risulta molto semplice l'aggiunta di più interfacce grafiche.

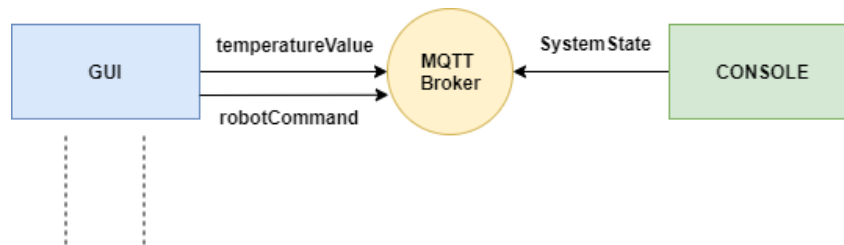


Figura 4.1: MQTT: comunicazione GUI e Console

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 é 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 conto 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 8.3 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 consolo 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 consolo 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.

TODO: PREMESSA: IL MODO è STATICO, NULLA CAMBIA. MA SAREBBE BUONA COSA PREVEDERE LA GESTIONE IN CASO DI ERRORI, OVVERO C'è UN OSTACOLO, IN PARTICOLARE CONSIDERAZIONE SU OSTACOLI DINAMICI

## 5.5 Gestione della mappa

Avendo ampliato il nostro sistema inserendo il *Robot retriever* un problema cruciale su cui discutere é 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 è 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 è 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

```

1  QActor robot_retriever_mind context ctxRobot {
2
3  Rules {
4    environment(notok).
5
6    eval(eq, X, X).    //since we have syntax limitations
7    doTheMove(M) :-
8      move(M1), !,
9      eval(eq,M,M1), !,
10     doTheFirstMove(M).
11
12     doTheFirstMove(w) :- retract( move(w) ),!.
13     doTheFirstMove(a) :- retract( move(a) ),!.
14     doTheFirstMove(d) :- retract( move(d) ),!.
15
16     homeReady :- curPos(0,0,D), bomb(_,_).
17     bombInRoom :- bomb(_,_).
18
19     nearBomb :- curPos(X,Y,_), eval(plus,X,1,R), bomb(R,Y).
20     nearBomb :- curPos(X,Y,_), eval(minus,X,1,R), bomb(R,Y).
21     nearBomb :- curPos(X,Y,_), eval(plus,Y,1,R), bomb(X,R).
22     nearBomb :- curPos(X,Y,_), eval(minus,Y,1,R), bomb(X,R).
23   }
24
25   // Set planner in retriever knowledge
26   State init initial [
27     javaRun it.unibo.planning.planUtil.initAI()
28   ] switchTo home
29
30   State home [
31
32   ] transition
33   stopAfter 6000000
34   whenEvent environment: environment(E) do
35     demo replaceRule(environment(X), environment(E)),
36     whenMsg map: map(M) do javaRun it.unibo.utils.updateStateOnRobot.loadMap(M),
37     whenMsg cmdReachBomb -> checkTemperatureAndRetrieve    // R-reachBag & R-TempOk
38     finally repeatPlan
39
40   State checkTemperatureAndRetrieve [          // R-TempOk
41     [ !? environment(ok) ] selfMsg cmdReachBomb: cmdReachBomb
42   ] transition
43   whenTime 100 -> home
44   whenMsg cmdReachBomb -> goToReachBomb
45
46   /*
47   * Retrieval business logic
48   */
49   State goToReachBomb [
50     println("STATE[goToReachBomb] ...");
51     javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-retrieving"
52       );
53     [ !? bomb(X,Y) ] javaRun it.unibo.planning.planUtil.setGoal(X,Y);

```

```

53     javaRun it.unibo.planning.planUtil.doPlan()
54 ] transition
55     whenTime 1000 -> doActions
56
57 State goToIdle [
58     println("RETRIEVER_MIND[goToIdle] ...");
59     [ !? nearBomb ] {
60         javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever -
61             retrieving");
62         delay 3000;
63         removeRule bomb(_,_);
64         selfMsg endAction: endAction
65     } else {
66         [ !? bombInRoom ] {
67             javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-idle");
68             forward robot_adapter -m robotCmd: robotCmd(blinkStop) // R-blinkLed (
69                 stop)
70         } else {
71             javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-idle-
72                 with-bomb");
73             forward robot_adapter -m robotCmd: robotCmd(blinkStop) // R-blinkLed (
74                 stop)
75         }
76     }
77 ] transition
78     whenTime 100 -> idle
79     whenMsg endAction -> goToHome
80
81 State goToHome [
82     println("STATE[goToHome] ...");
83     javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-returning")
84     ;
85     forward robot_adapter -m robotCmd: robotCmd(blinkStart) // R-blinkLed (
86         start)
87 ] switchTo backToHome
88
89 State idle [
90     // reach bomb solo se non ha bomba, go home in entrambi
91     [ !? bombInRoom ] selfMsg idleWhileRetrieving: idleWhileRetrieving
92     else selfMsg idleWhileReturning: idleWhileReturning
93 ] transition
94     stopAfter 6000000
95     whenMsg cmdStop -> idle,
96     whenMsg idleWhileRetrieving -> idleWhileRetrieving,
97     whenMsg idleWhileReturning -> idleWhileReturning
98
99 State idleWhileRetrieving [
100     javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-idle-
101         retrieving")
102 ] transition
103     stopAfter 6000000
104     whenMsg cmdStop -> idleWhileRetrieving,
105     whenMsg cmdReachBomb -> goToReachBomb,
106     whenMsg cmdGoHome -> goToHome // R-backHome
107
108 State idleWhileReturning [
109     javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("retriever-idle-
110         returning")
111 ] transition
112     stopAfter 6000000
113     whenMsg cmdStop -> idleWhileReturning,
114     whenMsg cmdGoHome -> goToHome // R-backHome
115
116 /*
117 * Planner
118 */
119 Plan doActions[
120     [ !? move(M) ] println( doActions_doingTheMove(M) );
121     [ not !? move(M) ] selfMsg endAction : endAction ;
122     [ !? move(M) ] selfMsg waitMoveCompleted: waitMoveCompleted;

```

```

115     [ !? doTheMove(M) ] forward robot_planner -m robotCmd: robotCmd(M)
116 ] transition
117 stopAfter 6000000
118 whenMsg cmdStop -> goToIdle,
119 whenMsg waitMoveCompleted -> waitMoveCompletedAnswer,
120 whenMsg endAction -> backToHome
121 finally repeatPlan
122
123 Plan waitMoveCompletedAnswer [
124
125 ] transition
126 stopAfter 6000000
127 whenMsg moveMsgCmdObstacle -> goToIdle,
128 whenMsg cmdStop -> goToIdle,
129 whenMsg moveMsgCmdDone -> handleCmdDone
130
131 Plan handleCmdDone [
132     printCurrentMessage;
133     onMsg moveMsgCmdDone: moveMsgCmdDone(X) ->
134         javaRun it.unibo.planning.planUtil.doMove(X); //update the map
135         javaRun it.unibo.planning.planUtil.showMap();
136         javaRun it.unibo.utils.updateStateOnConsole.updateMap()
137 ] transition
138 whenTime 100 -> doActions
139 whenMsg cmdStop -> goToIdle
140
141 Plan backToHome [
142     [ !? curPos(0,0,D) ]{
143         forward robot_adapter -m robotCmd: robotCmd(blinkStop); // R-blinkLed (stop)
144         println("AT HOME");
145         javaRun it.unibo.planning.planUtil.showMap();
146         javaRun it.unibo.utils.updateStateOnConsole.updateMap();
147         [ !? homeReady ] selfMsg cmdGoHome: cmdGoHome
148         else selfMsg endAction: endAction
149     }
150     else{
151         javaRun it.unibo.planning.planUtil.setGoal("0","0");
152         [ !? curPos(X,Y,D) ] println( backToHome(X,Y,D) );
153         javaRun it.unibo.planning.planUtil.doPlan()
154     }
155 ] transition
156 whenTime 100 -> doActions
157 whenMsg endAction -> terminate,
158 whenMsg cmdGoHome -> home,
159 whenMsg robotHomeAfterBomb -> home
160
161 State terminate [
162     println("STATE[terminate] retriever home!");
163     javaRun it.unibo.utils.updateStateOnConsole.updateRobotState("terminating")
164 ]
165
166 Plan handleError[ println("mind ERROR") ]
167
168 }

```

# Capitolo 6

## Sprint 6

### 6.1 Sistema distribuito

# Capitolo 7

## Il sistema

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

### 7.1 Architettura logica

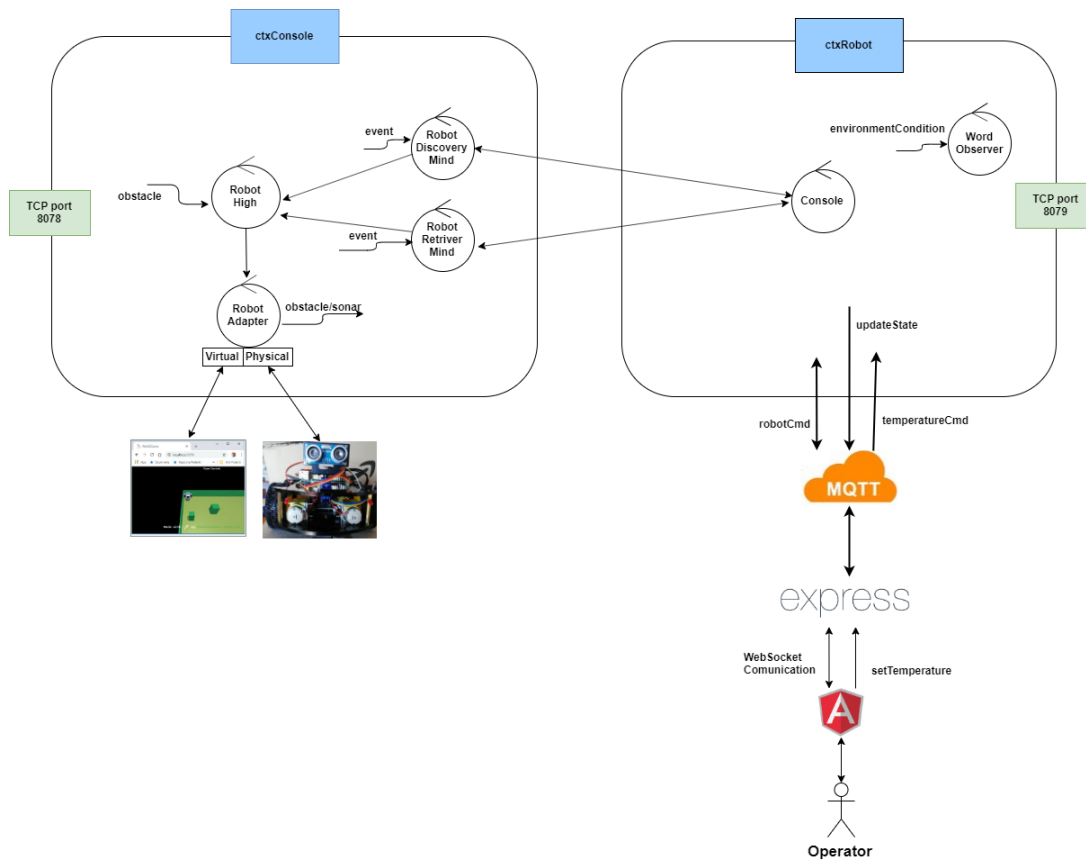


Figura 7.1: Architettura ddr system

### **7.1.1 Finite State Automaton (FSM)**

**Console**

**Word Observer**

**Robot Adapter**

**Robot Discovery Mind**

**Robot Retriever Mind**

comunicazione per messaggi



# Capitolo 8

## Tecnologie

### 8.1 QActor

### 8.2 Robot Virtuale

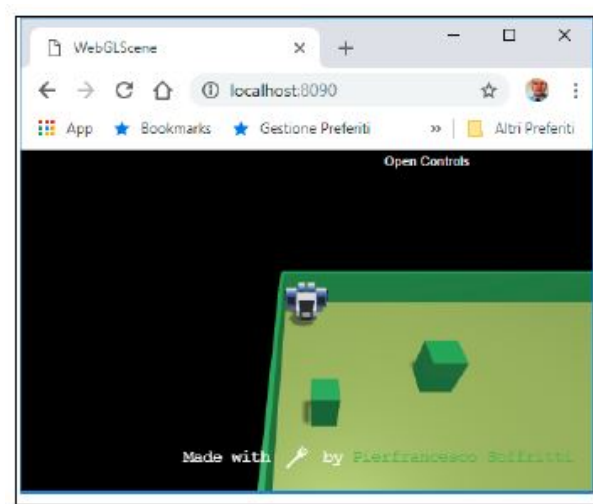


Figura 8.1: Soffritti: un simulatore virtuale

#### 8.2.1 Soffritti

### 8.3 Robot Fisico

#### 8.3.1 Arduino

### 8.4 Console

#### 8.4.1 Node

#### 8.4.2 Express

#### 8.4.3 Angular

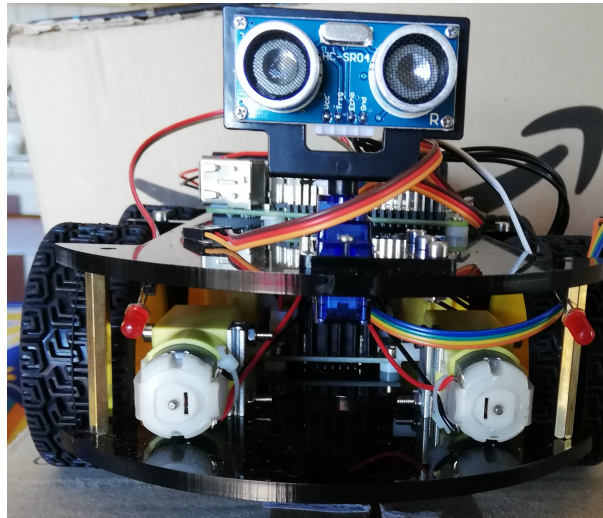


Figura 8.2: Robot fisico

### Robot console page

Robot current position in coordinates: ( 0, 0 )

Robot current movement: w

Robot current state: demo

explore home stop bomb bag

Set temperature

Change

### Hall

Temperature: 20 °C


Figura 8.3: Interfaccia utente, guidata dall'operatore umano

# Capitolo 9

## Note

NOTA: RICORDATI IN FONDO ALLA RELAZIONE DI AGGIUNGERE PARAGRAFO NOTE, DOVE SCRIVERE PROBLEMI IRRISOLVIBILI....

qA non permette attori dinamici

business logic sul robot (ragiona da analista)

problematica grosso modo: il robot riceve comando di partire ma deve valutare la temperatura e quindi verificare se le condizioni sono rispettate primo sprint ho la mappa, che se poi non va sti cazzi! soffritti con robot virtuale ????

strumenti flessibili di prototipazione (qui dispatch) punti : -  $\dot{}$  temperatura in qactor eval(ex, T, 35)- formalizzato dentro il robot.+

temperatura – parti dal più facile ma si preparato ad ampliare (chiedere di poter aggiornare la temperatura dinamicamente e il robot deve reagire mentre è in fase di explore)

eventi è qualcosa che la sorgente che emette l'informazione non sa chi è il destinatario. Quindi essendo che il robot deve inviare i dati dello stato alla sola console non è vero che è un evento ma è un dispatch. – requisito anche di SICUREZZA, voglio che solo la console conosce il mio stato. IL REQUISITO RICHIEDE MESSAGGIO QUINDI SI FA CON MESS.

meglio avere la logica incapsulata all'interno del robot [le condizioni di esercizio].

NOTE IMPLEMENTATIVA:

far vedere anche in che verso sta andando...

bottoni explore che si attiva solo quando la temp è inferiore a

tratta view la temp e se si muove o no.

robot-retrivier (insomma ha un sw diverso che gira al suo interno). quindi per il momento robot discovery - robot retriviel - console

bisogna chiedere al committente se il secondo robot deve partire senza calcolare variabili esterne (es. deve valutare la temperatura)

MQTT non va bene

debug complesso