

Software Engineering Project

Angelo Feraudo - Alessandro Staffolani

Alma Mater Studiorum – University of Bologna

viale Risorgimento 2, 40136 Bologna, Italy

angelo.feraudo@studio.unibo.it

alessandr.staffolan3@studio.unibo.it

1 Visione

Nella nostra vision per un moderno processo di sviluppo di sistemi software, si configura come essenziale un approccio model driven incrementale. Questo approccio infatti permette un'indipendenza dalle tecnologie realizzative e una minore sensibilità rispetto ad altri approcci. In particolare il nostro approccio sfrutta appieno la tecnologia fornita da **git**, la quale ci ha permesso di separare il carico di lavoro creando un branch per ogni requisito. Questo ci ha consentito di effettuare di volta in volta dei test sui singoli requisiti in maniera incrementale. Inoltre un altro obiettivo fondamentale della nostra visione è la creazione di un sistema che rispetti quanto possibile l'architettura esagonale, garantendo una grande modularità del codice.

2 Requisiti

In a home of a given city (e.g. Bologna), a **ddr** robot is used to clean the floor of a room (**R-FloorClean**). The floor in the room is a flat floor of solid material and is equipped with two sonars, named **sonar1** and **sonar2** as shown in the picture (sonar1 is that at the top). The initial position (**start-point**) of the robot is detected by **sonar1**, while the final position (**end-point**) is detected by **sonar2**.

1. **R-Start:** an authorized user has sent a START command by using a human GUI interface (console) running on a conventional PC or on a smart device (Android).
2. **R-TempOk:** the value temperature of the city is not higher than a prefixed value (e.g. 25 degrees Celsius).
3. **R-TimeOk:** the current clock time is within a given interval (e.g. between 7 a.m and 10 a.m).
4. While the robot is working:
 - it must blink a Led put on it, if the robot is a real robot (**R-BlinkLed**).

- it must blink a Led Hue Lamp available in the house, if the robot is a virtual robot (**R-BlinkHue**).
- it must avoid fixed obstacles (e.g. furniture) present in the room (**R-AvoidFix**) and/or mobile obstacles like balls, cats, etc. (**R-AvoidMobile**).

5. Moreover, the robot must stop its activity when one of the following conditions apply:

- **R-Stop**: an authorized user has sent a STOP command by using the console.
- **R-TempKo**: the value temperature of the city becomes higher than the prefixed value.
- **R-TimeKo**: the current clock time is beyond the given interval.
- **R-Obstacle**: the robot has found an obstacle that it is unable to avoid.
- **R-End**: the robot has finished its work.

6. During its work, the robot can **optionally**

- **R-Map**: build a map of the room floor with the position of the fixed obstacles. Once built, this map can be used to define a plan for an (optimal) path from the start-point to the end-point.

3 Analisi dei requisiti

Da un'attenta analisi dei requisiti abbiamo dedotto che sono presenti un robot e due sonar che indicano rispettivamente la posizione iniziale e finale. Lo scopo finale di questo robot sarà quello di pulire completamente il pavimento di una stanza.

Nei prossimi paragrafi verranno analizzati i singoli requisiti posti dal cliente.

3.1 Analisi requisito 1

R-Start: *an authorized user has sent a START command by using a human GUI interface (console) running on a conventional PC or on a smart device (Android).*

La prima cosa che si deduce leggendo questo requisito è la necessità di un nuovo componente chiamato *console* (GUI interface) che deve essere accessibile solo ad un **utente autorizzato** dai seguenti dispositivi:

- PC;
- smart device

Al termine dell'analisi di questo requisito chiediamo al committente:

1. Cosa si intende per utente autorizzato?

Dopo un'interazione con il committente si è stabilito che un utente per essere autorizzato deve avere a disposizione un **username** e una **password**.

3.1.1 Analisi requisito 1: Console .

La **console** non è altro che una *graphic user interface* che permette all'utente di avviare il robot per la pulizia della stanza. Dal requisito è implicito che, oltre a essere presente un comando di **START** (per l'avvio della pulizia della stanza), sarà presente un comando di **STOP** per fermare il processo di pulizia del robot.

3.2 Analisi requisito 2

R-TempOk: *the value temperature of the city is not higher than a prefixed value (e.g. 25 degrees Celsius).*

Dall'analisi di questo requisito si evince che è necessario ottenere la temperatura in gradi Celsius di una città e che quest'ultima non debba superare un certo valore prefissato (ad esempio di 25 gradi). Al termine dell'analisi di questo requisito chiediamo al committente:

1. La città da tenere in considerazione è quella relativa alla posizione del robot o quella relativa alla posizione dell'utente autorizzato?

Dopo un'interazione con il committente abbiamo stabilito che la temperatura è relativa alla città del robot.

3.3 Analisi requisito 3

R-TimeOk: *the current clock time is within a given interval (e.g. between 7 a.m and 10 a.m).*

Dall'analisi di questo requisito si deduce che è necessario stabilire l'ora attuale e che quest'ultima debba rientrare in un determinato intervallo (ad esempio 7:00 e 10:00), affinchè il robot possa muoversi. Al termine dell'analisi di questo requisito chiediamo al committente:

1. L'orario deve essere newtoniano o relativistico?
2. L'orario deve essere relativo al fuso orario del robot o a quello dell'utente autorizzato?

Dopo un'interazione con il committente abbiamo stabilito che l'orario deve essere **newtoniano** e deve essere relativo al fuso orario del **robot**.

3.4 Analisi requisito 4

While the robot is working:

- it must blink a Led put on it, if the robot is a real robot (**R-BlinkLed**).
- it must blink a Led Hue Lamp available in the house, if the robot is a virtual robot (**R-BlinkHue**).
- it must avoid fixed obstacles (e.g. furniture) present in the room (**R-AvoidFix**) and/or mobile obstacles like balls, cats, etc. (**R-AvoidMobile**).

Dopo un'analisi attenta di questo requisito, si introduce un nuovo componente, il **led**, che dovrà lampeggiare durante il movimento del robot.

Da questo requisito inoltre si deduce che sono presenti due tipologie di robot:

1. un **robot virtuale**;
2. un **robot fisico**

Altri particolari importanti di questo requisito sono i seguenti:

- nel caso in cui si tratti del robot virutale, a lampeggiare non deve essere un led ma una **Led Hue Lamp** disponibile all'interno della casa;
- nel caso si tratti del robot fisico, a lampeggiare deve essere un **led** posto su di esso;
- entrambi i robot dovranno inoltre evitare gli ostacoli presenti nell'ambiente (virtuale o reale), sia che essi siano **fissi** o **mobili**.

3.4.1 Analisi requisito 4: LED .

In elettronica il LED (Light Emitting Diode) o diodo a emissione di luce è un dispositivo optoelettronico che sfrutta la capacità di alcuni materiali semiconduttori di produrre fotoni attraverso un fenomeno di emissione spontanea.



Figura 1: Led

Tale led sarà posto direttamente sul **real robot**.

3.4.2 Analisi requisito 4: Led Hue Lamp .

La lampada HUE è una lampada controllabile remotamente. In particolare per comunicare usa *Zigbee Lighting protocol* e può essere controllata attraverso un'applicazione smartphone mediante connessione wi-fi.

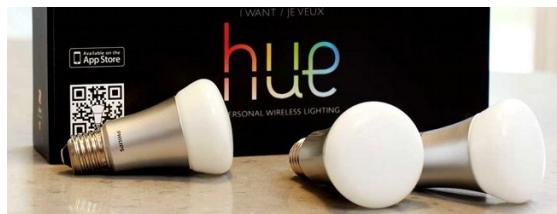


Figura 2: HUE

3.5 Analisi requisito aggiuntivo

Moreover, the robot must stop its activity when one of the following conditions apply:

- **R-Stop:** an authorized user has sent a STOP command by using the console.
- **R-TempKo:** the value temperature of the city becomes higher than the prefixed value.
- **R-TimeKo:** the current clock time is beyond the given interval.
- **R-Obstacle:** the robot has found an obstacle that it is unable to avoid.
- **R-End:** the robot has finished its work.

Questo requisito è aggiuntivo in quanto, come già si poteva dedurre dai requisiti precedenti, il committente puntualizza quando il robot (virtuale o fisico) deve fermarsi.

In particolare:

- quando l'utente autenticato invia un comando di **stop** (requisito 1);
- quando il valore della **temperatura** supera un certo valore prefissato (requisito 2);
- quando l'**orario** è al di fuori di un intervallo prefissato (requisito 3);
- quando il robot si trova davanti ad un **ostacolo** che non riesce ad evitare (requisito 4);
- quando il robot **termina** il lavoro.

3.6 Analisi requisito 5

*During its work, the robot can **optionally***

- **R-Map:** build a map of the room floor with the position of the fixed obstacles.
Once built, this map can be used to define a plan for an (optimal) path from the start-point to the end-point.

Da questo requisito si deduce che il robot deve tener traccia del suo movimento per poter disegnare la mappa della stanza, tenendo conto di eventuali

ostacoli fissi. Inoltre, una volta che il robot è riuscito a costruire la mappa della stanza, dovrà pianificare il percorso ottimo da effettuare per raggiungere l'*end-point* a partire dallo *start-point*.

4 Requisito 1

4.1 Analisi del problema

Dall'analisi del *requisito uno* emerge che la struttura del sistema avrà di sicuro due componenti:

- **robot**;
- **console**.

La loro **interazione** avviene mediante lo **scambio di messaggi**, preferiti rispetto agli eventi, in quanto nel modello a scambio di messaggi c'è una bassa probabilità di perdita dell'informazione. Ciò dipende dal fatto che tutti i messaggi verranno messi all'interno di una coda nel caso in cui il ricevente sia già occupato. Inoltre gli eventi potrebbero avere più di un destinatario, ma non è questo il caso, in quanto si presume che la console lavori con un robot alla volta. Dall'analisi dei requisiti si deduce che la console lavorerà in un **contesto separato** rispetto a quello del robot, in quanto sarà possibile che entrambi lavorino su dei nodi separati.

Per realizzare un prototipo utilizzeremo un linguaggio custom realizzato dalla nostra software house chiamato **QActor**.

4.1.1 Analisi del problema: Console .

Prima di realizzare un prototipo della console, si assume che essa abbia già verificato l'identità dell'utente e che quindi solo chi autorizzato possa inviare i comandi **startBot** e **stopBot**. Utilizzando il linguaggio custom, realizziamo un prototipo che simulerà il comportamento della console e quindi i suoi relativi comandi.

Tale prototipo ci aiuterà ad effettuare un test sul funzionamento corretto dello scambio di messaggi, in questo caso **consoleGui(CMD)**, tra robot e console.

```

@QActor consoleanalysis context ctxConsoleAnalysis -g cyan {
    Plan init normal [
        println("Console ready");
        delay 5000;
        forward robotanalysis -m moveRobot : usercmd( consoleGui( startBot ) );
        delay 1000;
        forward robotanalysis -m moveRobot : usercmd( consoleGui( stopBot ) )
    ]
}

```

Figura 3: *QActor* console

4.1.2 Analisi del problema: Robot .

Per avere un prototipo completo che riesca a simulare il requisito uno in maniera completa, abbiamo creato un ulteriore *QActor* che vada a simulare il comportamento del robot.

```

QActor robotanalysis context ctxRobotAnalysis {
    Plan init normal [
        println("Robot ready")
    ]
    switchTo waitForCmd

    Plan waitForCmd[ ]
    transition stopAfter 3600000 //1h
    whenMsg moveRobot -> execMove
    finally repeatPlan

    Plan execMove resumeLastPlan [
        printCurrentMessage;
        onMsg moveRobot : usercmd( consoleGui( startBot ) ) -> {
            println("Inizio a spazzare")
        };
        onMsg moveRobot : usercmd( consoleGui( stopBot ) ) -> {
            println("Termino di spazzare")
        }
    ]
}

```

Figura 4: *QActor* robot

In particolare:

- ogni volta che il robot riceverà un messaggio di **start**, stamperà la seguente stringa: *Inizio a spazzare*
- ogni volta che il robot riceverà un messaggio di **stop**, stamperà la seguente stringa: *Termino di spazzare*

Il test, per essere il più fedele possibile all'ambiente reale, si effettuerà eseguendo il protipo della console e quello del robot su contesti separati, in modo da poter anche simulare un **ambiente distribuito**.

4.2 Progettazione

In fase di progettazione per riuscire ad avere un'idea di come poter procedere poi nell'implementazione, si è deciso di realizzare una struttura come quella illustrata nella figura 5.

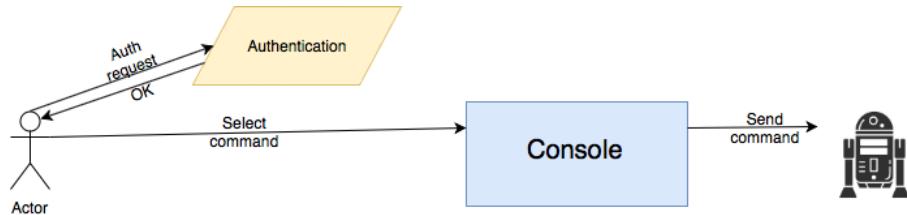


Figura 5: Struttura iniziale

Dal grafico si può notare come non viene fatta una distinzione tra *robot reale* e *robot virtuale*, in quanto i comandi inviati dalla console verranno "catturati" da entrambi e, poi interpretati dal robot attualmente attivo.

Per realizzare un comportamento del genere si utilizza una tecnologia che segue il modello **publish/subscribe**: in questo modello, mittenti e destinatari di messaggi dialogano attraverso un tramite, detto **dispatcher** o **broker**. Il mittente di un messaggio (detto publisher) non deve essere consapevole dell'identità dei destinatari (detti **subscriber**); esso si limita a "pubblicare" il proprio messaggio al dispatcher. La tecnologia scelta per realizzare questo modello è **MQTT** (Message Queue Telemetry Transport).

Nella figura 6 è mostrato come, a grandi linee, dovrebbe funzionare tale modello.

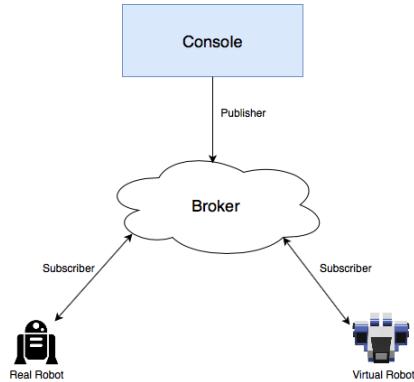


Figura 6: Modello publish/subscribe

4.2.1 Progettazione: Robot .

Dall'analisi dei requisiti ci siamo accorti che non esiste una sola versione del robot, ma due:

- una **Virtuale** (virtual robot);
- una **Reale** (real robot).

Quindi prima di procedere con la progettazione del robot è stata fatta la seguente considerazione: visto che le versioni sono due, ci conviene fare già a partire dalla progettazione del requisito uno, un modello che catturi gli aspetti essenziali di entrambi.

Per questo motivo nelle seguenti sezioni analizzeremo i due modelli corrispondenti alle due versioni.

4.2.2 Progettazione: Adapter Real Robot .

Il robot sarà realizzato mediante le seguenti componenti (**Fig. 7**):

- 1 x Raspberry pi 3;
- 1 x Bridge motor (L298N) che permetterà la gestione dei due motori (avanti e indietro);
- 2 x Gear Motor;

- 1 x Sesore ultrasonico (HC-SR04);
- 1 x green led;
- 1 x power bank.

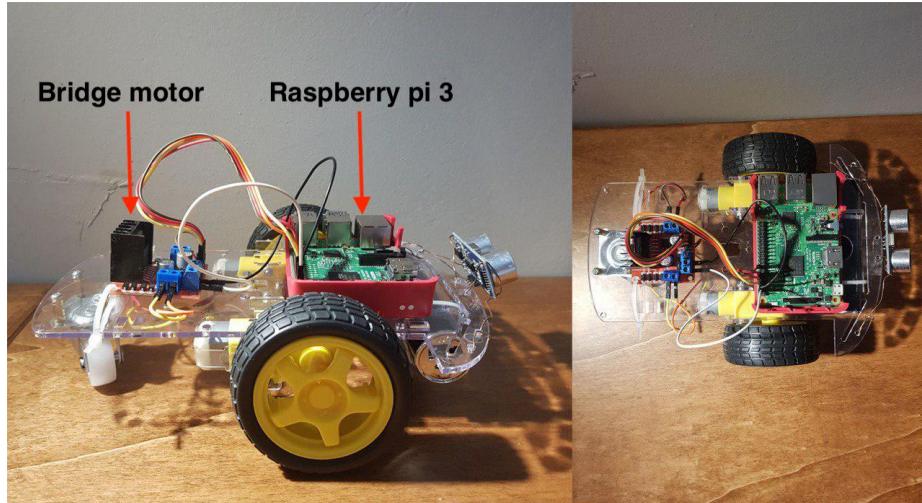


Figura 7: Real Robot

Il robot è una componente IoT del sistema, questo significa che deve essere in grado di connettersi alla rete e di eseguire dei compiti inviati in remoto dalla console. Per diventare una componente IOT, il **Real Robot** usa una **raspberry pi 3** che, oltre a essere fondamentale per la gestione del movimento del robot, sarà fondamentale per il collegamento del robot al sistema distribuito.

Le restanti componenti saranno utili per soddisfare i requisiti successivi; avendo adottato un procedimento incrementale, queste verranno analizzate quando ce ne sarà bisogno.

Per rendere il prototipo funzionante il real robot verrà modellato, attraverso il linguaggio custom (*QActor*) fornito dalla nostra software house, come mostrato in figura.

Dal modello (Fig. 8) si può notare come il robot sia sempre in attesa di un

```

QActor robotexecutor context ctxRobot -pubsub {
    Plan init normal [
        println("Robot ready")
    ]
    switchTo waitForCmd

    Plan waitForCmd[ ]
        transition stopAfter 3600000 //1h
            whenMsg moveRobot -> execMove
        finally repeatPlan

    Plan execMove resumeLastPlan [
        printCurrentMessage;
        onMsg moveRobot : usercmd( robotgui(h(X)) ) ->
            javaOp "customExecute(\"python3 executor.py h\")";
        onMsg moveRobot : usercmd( robotgui(w(X)) ) ->
            javaOp "customExecute(\"python3 executor.py w\")";
        onMsg moveRobot : usercmd( robotgui(s(X)) ) ->
            javaOp "customExecute(\"python3 executor.py s\")";
        onMsg moveRobot : usercmd( robotgui(a(X)) ) ->
            javaOp "customExecute(\"python3 executor.py a\")";
        onMsg moveRobot : usercmd( robotgui(d(X)) ) ->
            javaOp "customExecute(\"python3 executor.py d\")"
    ]
}

```

Figura 8: Modello Robot Reale

messaggio (**moveRobot**) che racchiude la mossa che il robot dovrà effettuare. Tale messaggio è analogo al messaggio (**consoleGui**) utilizzato in fase di testing dei messaggi nell'analisi del problema.

Da notare che ogni qual volta che viene ricevuto un messaggio viene eseguita la seguente riga:

```
javaOp "customExecute( "python3 executor.py h ");
```

Questa viene aggiunta soltanto dopo aver testato che i messaggi arrivassero correttamente e dopo aver testato che lo script *python*, che verrà analizzato in fase di implementazione, funzionasse correttamente. Questo modello verrà poi eseguito direttamente sulla raspberry e avrà la funzione di **adapter**, in quanto farà corrispondere ai comandi inviati dalla console l'esecuzione dello script python, di cui parleremo in fase di implementazione.

4.2.3 Progettazione: Adapter Virtual Robot .

Il virtual robot è un'applicazione web fornita dalla software house, che ci sarà utile sia per simulare i comportamenti del robot fisico in un ambiente virtuale,

sia per effettuare operazioni di testing, che simulino l'ambiente reale.

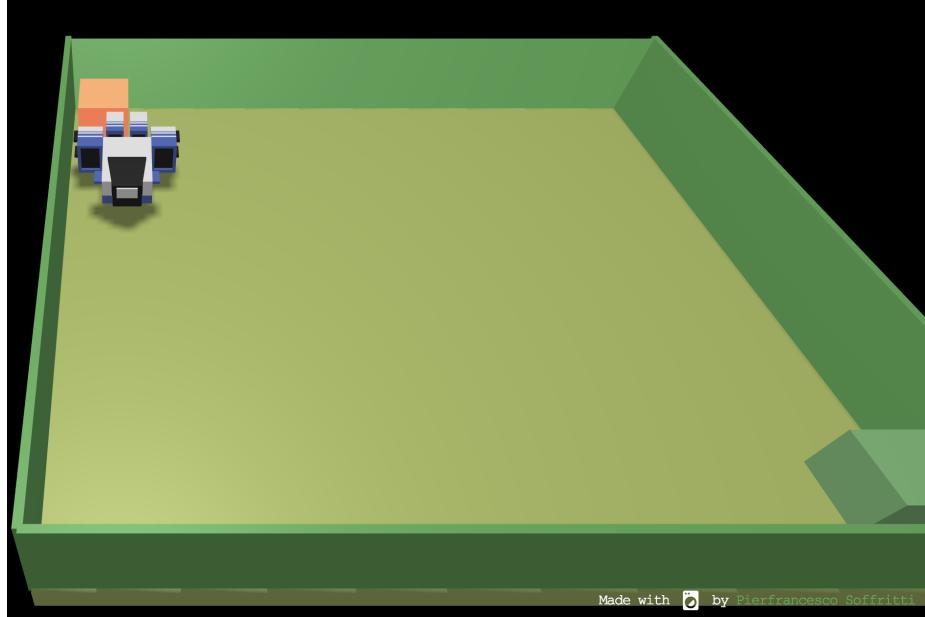


Figura 9: Virtual Robot

Come si può notare dalla figura, sono presenti due sonar (start-point e end-point) che delimitano la stanza in cui il virtual robot potrà muoversi. Da ciò ne consegue che la stanza simulabile potrà essere o un quadrato o un rettangolo.

Inoltre in quest'ambiente virtuale, attraverso degli opportuni file messi a disposizione dal componente, sarà possibile inserire ostacoli fissi (tavoli, armadi, sedie, ecc.) e ostacoli mobili (animali, persone, palloni, ecc.) per permettere una simulazione più accurata. Come nel caso del real robot, anche quest'ultimo è dotato di un sonar in grado di rilevare gli ostacoli. Infatti, ogni qualvolta che verrà incontrato un ostacolo, il virtual robot pubblicherà attraverso un **software Java** (**clientTCP** fornito dalla software house) sul *broker* di **mqtt** il seguente messaggio:

```
msg(sonarDetect,event,virtualrobot_executorctrl,
no-
ne,sonarDetect(wallLeft,soffritti),22)
```

Per rendere il prototipo funzionante il robot virtuale è stato modellato come in figura 10.

```
QActor virtualrobotexecutor context ctxVirtualRobot -pubsub {
    Plan init normal [
        init it.unibo.utils.clientTcp.initClientConn("localhost","8999");
        delay 1000;
        javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'turnRight', 'arg': 800 }"); // necessario per non farlo andare contro il sonar appena parte
        println("Virtual robot ready")
    ] switchTo waitForCmd
    Plan waitForCmd[ ]
    transition stopAfter 3600000 //1h
    whenMsg moveRobot → execMove
    finally repeatPlan
    Plan execMove resumeLastPlan [
        printCurrentMessage;
        onMsg moveRobot : usercmd( robotgui(hX) ) →
            javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'alarm' }");
        onMsg moveRobot : usercmd( robotgui(wX) ) →
            javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveForward', 'arg': -1 }");
        onMsg moveRobot : usercmd( robotgui(sX) ) →
            javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'moveBackward', 'arg': -1 }");
        onMsg moveRobot : usercmd( robotgui(aX) ) →
            javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'turnLeft', 'arg': 800 }");
        onMsg moveRobot : usercmd( robotgui(rX) ) →
            javaRun it.unibo.utils.clientTcp.sendMsg("{ 'type': 'turnRight', 'arg': 800 }")
    ]
}
```

Figura 10: Modello Robot Virtuale

Come nel caso del robot reale, anche in questo caso il messaggio usato è **moveRobot** e, come nel caso precedente, prima dell'invio del messaggio al robot virtuale fornito dalla software house sono stati effettuati dei test sul corretto funzionamento dei messaggi. Prima di andare a vedere come sono stati progettati internamente il blocco **console** e il blocco **authentication**, verranno introdotte alcune tecnologie utilizzate.

4.2.4 Progettazione: Tecnologie .

La console verrà realizzata mediante **ReactJS** una tecnologia recente, scritta in JavaScript, che permette di realizzare, lato frontend, una '*One page application*', ovvero un'applicazione web che carica la pagina solo la prima volta che l'utente la richiede e poi aggiorna il contenuto in maniera dinamica andando ad iniettare i componenti senza bisogno di ricaricare la stessa. La scelta di questa tecnologia è motivata dal fatto che il trend attuale è proprio quello di realizzare

queste cosidette '*One page application*' perchè garantiscono una migliore **user-experience**.

Come vedremo successivamente, questa applicazione web necessita di un **backend** in grado di comunicare con il robot e in grado di memorizzare su di un database i dati dell'utente per verificare l'accesso.

Per la realizzazione del backend abbiamo scelto un'altra tecnologia molto utilizzata, **NodeJS**, una piattaforma scritta in JavaScript che sfrutta il motore di **Google Chrome V8** per la realizzazione di server web con modello **asincrono**, detto *event-driven*. Si tratta di un modello in cui, quando si richiedono operazioni "lente" - come ad esempio le operazioni di I/O (input/output) -, il sistema non resta in attesa del loro completamento, ma continua ad eseguire il codice sottostante; solo quando tale operazione termina si esegue una cosiddetta '*callback*' con le istruzioni da eseguire.

Per la gestione del database la tecnologia che è stata scelta è **MongoDB**, ovvero un database non relazionale (**NoSQL**) che si allontana dalla struttura tradizionale basata su tabelle dei database relazionali a favore di documenti in stile JSON con schema dinamico. Considerando le operazioni che devono essere effettuate e le tecnologie utilizzate per la realizzazione del backend, questa risulta essere la migliore alternativa in termini di performance, costi e anche supporto da parte della community.

4.2.5 Progettazione: Authentication .

L'autenticazione, successiva alla registrazione dell'utente mediante la scelta di **username** e **password**, si compone dei seguenti passi:

1. L'utente inserisce, nella pagina di login della web application, '**username** e **password**;
2. l'applicazione richiede al web server di verificare l'esistenza dell'username e la validità della password inseriti dall'utente;
3. se i dati sono validi, il web server restituisce un token di accesso valido per un'ora e che permette all'utente di accedere alla console.

4.2.6 Progettazione: Console .

La console si compone di tre componenti:

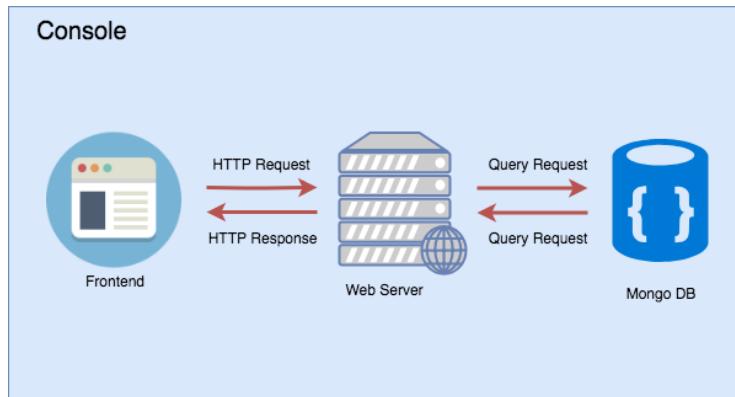


Figura 11: Struttura console

1. **Frontend** realizzato con **ReactJS**, interfaccia attraverso cui l'utente potrà interagire con il sistema;
2. **Web Server** realizzato con **NodeJS**, che eseguirà le richieste effettuate dal frontend;
3. **Data Base** realizzato utilizzando **MongoDB**, per memorizzare le informazioni relative all'utente autenticato.

4.3 Implementazione

4.3.1 Implementazione: Authentication .

A livello implementativo l'autenticazione è stata realizzata:

- lato **frontend** mediante un componente React contenente il form di login, che prenderà i dati inseriti dall'utente e li invierà al webserver tramite richiesta HTTP;
- lato **backend** mediante controllo dei dati inseriti dall'utente. A questo punto se i dati inseriti sono validi si controlla se l'username è presente all'interno del database e che la password sia quella corretta. Se tutto è andato a buon fine verrà restituito un **JSON Web Token** (JWT), che permetterà all'utente di rimanere autenticato per un'ora.

Il JWT è una stringa codificata in **BASE64** composta da 3 componenti:

- La prima contiene informazioni sull'algoritmo usato per generare la firma di verifica (Verify Signature);
- La seconda parte contiene il payload, ovvero le informazioni relative all'utente e al periodo di validità che si vogliono ottenere quando si decodifica il token;
- La terza parte contiene la Verify Signature usata poi per controllare l'effettiva autenticità del token.

4.3.2 Implementazione: Console .

Nell'implementazione della console abbiamo deciso, per completezza, di aggiungere, oltre al comando start e stop, dei comandi che permetteranno all'utente di comandare manualmente il robot.

In particolare i comandi sono i seguenti:

- **Forward**: permette al robot di andare avanti;
- **Backward**: permette al robot di andare indietro;
- **Left**: permette al robot di girarsi a sinistra;

```
exports.login = (req, res, next) => {
    let requested_user = req.body.user;
    if (abstractController.body_is_valid(req, res, next, requested_user)) {
        // data are correct check if user exist
        User.findOne({username: requested_user.username})
            .then(user => {
                if (user != null) {
                    // if user exist check password
                    user.comparePassword(requested_user.password)
                        .then(result => {
                            if (result) {
                                // password correct return username and authenticate true
                                abstractController.return_request(req, res, next, {
                                    authenticate: true,
                                    userId: user.id,
                                    username: user.username,
                                    role: user.role,
                                    token: jsonWebToken.generateToken(user),
                                })
                            } else {
                                let errorPayload = {
                                    errors: {
                                        "user.password": {
                                            param: "password",
                                            value: requested_user.password,
                                            command: 'Password is not correct'
                                        }
                                    },
                                    requestObject: requested_user
                                };
                                abstractController.return_bad_request(req, res, next, errorPayload);
                            }
                        }).catch(err => next(err))
                } else {
                    let errorPayload = {
                        errors: {
                            "user.username": {
                                param: "username",
                                value: requested_user.username,
                                command: 'Username not exist'
                            }
                        },
                        requestObject: requested_user
                    };
                    abstractController.return_bad_request(req, res, next, errorPayload);
                }
            }).catch(err => next(err));
};
```

Figura 12: Implementazione Login

- **Right**: permette al robot di girarsi a destra;
 - **Stop**: permette al robot di fermarsi (corrisponde al comando di STOP richiesto nei requisiti);
 - **Allow Autopilot**: permette al robot di iniziare la pulizia della stanza (corrisponde al comando di START richiesto nei requisiti);
-

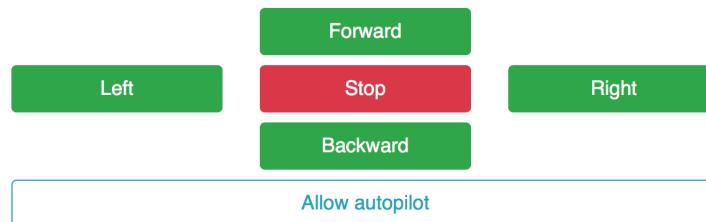


Figura 13: Console

L’ultimo comando non verrà implementato in questo momento, ma successivamente, quando si inizierà a pensare sul come il robot debba pulire e costruire la mappa della stanza (vedi Requisito 5).

Come preannunciato in fase di progettazione, ogni qual volta l’utente autorizzato cliccherà su un pulsante messo a disposizione dal frontend, verrà inviata una richiesta **HTTP** al server che a sua volta (come mostrato in figura 6) pubblicherà un messaggio sul broker di mqtt, che indicherà il movimento che il robot dovrà effettuare. In questo modo tutti i robot "iscritti" al broker riceveranno il comando inviato dall’utente.

Degno di nota è il fatto che il web server sia **RESTful**. Infatti, ogni volta che esso riceve una richiesta non effettua mai il render di una pagina HTML come risposta, ma semplicemente ritorna una stringa JSON, dove segnala che tutto è andato a buon fine.

La console così implementata ce la porteremo avanti fino al termine dell’inte-

ro progetto, aggiungendo poi il comando relativo alla pulizia automatica della stanza.

4.3.3 Implementazione: Robot .

Avendo già definito in fase di progettazione il modello del **Virtual Robot** e essendo già pronto per l'uso visto che viene fornito dalla nostra software house, tratteremo nell'implementazione soltanto ciò che permette il movimento del **Real Robot**.

Lo script python sfrutta, per rendere possibile il movimento del robot, la tecnologia **GPIO** messa a disposizione dalla raspberry.

```
1 import RPi.GPIO as GPIO
2 import time
3 import sys
4
5 GPIO.setmode(GPIO.BCM)
6
7 BL = 14
8 FL = 15
9 FR = 18
10 BR = 23
11 TRIG = 24
12 ECHO = 25
13
14 GPIO.setwarnings(False)
15 GPIO.setup(FR, GPIO.OUT)
16 GPIO.setup(FL, GPIO.OUT)
17 GPIO.setup(BL, GPIO.OUT)
18 GPIO.setup(BR, GPIO.OUT)
19
20
21
22 def move_forward():
23     GPIO.output(FR,True)
24     GPIO.output(FL,True)
25
26 def move_right():
27     GPIO.output(FR,True)
28     time.sleep(1)
29     GPIO.output(FR,False)
30     GPIO.cleanup()
31
32 def move_left():
33     GPIO.output(FL,True)
34     time.sleep(1)
35     GPIO.output(FL,False)
36     GPIO.cleanup()
37
38 def move_backward():
39     GPIO.output(BL,True)
40     GPIO.output(BR,True)
```

Figura 14: Script python Parte 1

Nella prima parte dello script viene stabilito in che modo utilizzare i PIN (**GPIO.BCM**, tenendo in considerazione la documentazione fornita dalla raspberry) e successivamente quali PIN utilizzare (da notare che i PIN **TRIG** e **ECHO** non verranno usati in questa versione dello script). Una volta che sono

stati stabiliti quali PIN verranno utilizzati si stabilisce come utilizzarli: **INPUT** o **OUTPUT**. Nel nostro caso ci serviranno solo PIN impostati su **OUTPUT**. Fatto questo vengono definite più funzioni che definiscono i movimenti che il robot può effettuare. In questa prima versione i movimenti di rotazione non sono ancora precisi, tuttavia riescono a darci un movimento accettabile per un primo prototipo.

```

42 def stop():
43     GPIO.output(FR,False)
44     GPIO.output(FL,False)
45     GPIO.output(BL,False)
46     GPIO.output(BR,False)
47     GPIO.cleanup()
48
49
50 if sys.argv[1] == "W" or sys.argv[1] == "w":
51     move_forward()
52
53 elif sys.argv[1] == "S" or sys.argv[1] == "s":
54     move_backward()
55
56 elif sys.argv[1] == "D" or sys.argv[1] == "d":
57     move_right()
58
59 elif sys.argv[1] == "A" or sys.argv[1] == "a":
60     move_left()
61
62 elif sys.argv[1] == "H" or sys.argv[1] == "h":
63     stop()
64
65 elif sys.argv[1] == "P" or sys.argv[1] == "p":
66     autopilot(30)
67 else:
68     print("Not recognized command!")
69

```

Figura 15: Script python Parte 2

Al termine di questa implementazione abbiamo un primo prototipo funzionante da mostrare al committente.

5 Requisito 2

5.1 Analisi del problema

L'analisi del *requisito 2* introduce un vincolo al nostro sistema distribuito, dal momento che - tenendo in considerazione anche l'analisi del requisito aggiuntivo 3.5 - il **Robot** potrà muoversi solo se il valore della temperatura è al di sotto di un certo valore (ad esempio 25 gradi Celsius).

Per riuscire ad effettuare la valutazione di questo vincolo bisogna fare delle modifiche a livello strutturale. Infatti risulterà più conveniente introdurre una nuova componente che si occupi soltanto della verifica dei vincoli. In questo modo riusciremo a separare la **logica** dall'esecuzione dei comandi veri e propri.

L'aggiunta di questa nuova componente porterà delle modifiche anche nella console, in quanto quest'ultima non dovrà più comunicare direttamente con il robot, ma dovrà farlo con la nuova componente introdotta, la quale stabilirà se sarà possibile o meno effettuare l'azione indicata.

Di seguito verrà creato un prototipo per effettuare dei test sul funzionamento dei vincoli solo a livello locale.

5.1.1 Analisi del problema: Mind .

La **Mind**, come dice la parola stessa, rappresenterà la mente del robot, in quanto dovrà stabilire se il robot potrà effettuare un movimento in base a dei vincoli che sono posti all'interno della base di conoscenza.

In questa fase di analisi del problema ci concentreremo principalmente su come realizzare la base di conoscenza, realizzando una prima modellazione della nuova componente e sapendo che la definizione della base di conoscenza di quest'ultima avviene mediante un linguaggio di programmazione logica(**Prolog**).

Nella base di conoscenza di questo modello viene introdotto il controllo del vincolo relativo alla temperatura, che abbiamo chiamato **checkTemperatu-re(VALUE)**. Questo controllo restituirà un valore, *True* o *False*, in base al

```

QActor mindrobotanalysis context ctxRobotAnalysis {
    Rules {
        eval( let, X, X ), // lower equal than implementation using worldTheory.pl in src-more/it/unibo/mindrobot/
        eval( let, X, V ) :- eval( lt, X, V ) .
        maxTemperature(25),
        currentTemperature(12),
        checkTemperature(cold),
        maxTemperature(MAX),
        currentTemperature(CURRENT),
        eval(let, CURRENT, MAX), !.
        checkTemperature(hot),
        maxTemperature(MAX),
        currentTemperature(CURRENT),
        eval(gt, CURRENT, MAX), !.
    }
    Plan init normal [
        println("Mind robot ready")
    ] switchTo waitPlan
    Plan waitPlan[
        transition stopAfter 3600000 //ih
        event constraint -> handleEvent,
        whenever moveRobot -> handleMsg
        finally repeatPlan
    ]
    Plan handleEvent resumeLastPlan [
        printCurrentEvent
        onEvent constraint : constraint(temp, V) -> ReplaceRule currentTemperature(X) with currentTemperature(V);
        [ !? checkTemperature(cold) ] forward robottexecutoranalysis -m execMoveRobot : usercmd( consoleGui( stopBot ) )
    ]
    Plan handleMsg resumeLastPlan [
        printCurrentMessage
        onMsg moveRobot : usercmd( consoleGui( startBot ) ) ->{
            [ !? checkTemperature(cold) ] forward robottexecutoranalysis -m execMoveRobot : usercmd( consoleGui( startBot ) )
            else println("Too hot to work")
        };
        onMsg moveRobot : usercmd( consoleGui( stopBot ) ) ->{
            [ !? checkTemperature(cold) ] forward robottexecutoranalysis -m execMoveRobot : usercmd( consoleGui( stopBot ) )
            else println("Too hot to work")
        }
    ]
}

```

Figura 16: Modello Mind Robot

fatto che il vincolo sia soddisfatto o meno.

Come già analizzato in precedenza, in questo caso la console non comunicherà direttamente con il robot, ma piuttosto con la mind; sarà compito di quest'ultima controllare, prima di inoltrare il messaggio al robot, se la temperatura è al di sotto della soglia indicata. Da notare che nel modello è stato introdotto il seguente evento:

Event constraint : constraint(CONSTRAINT, VALUE)

Figura 17: Event

Esso servirà per modificare il valore della temperatura, considerando il fatto che la temperatura di un luogo potrebbe cambiare.

La restante parte di modellazione è uguale a quella introdotta nell'analisi del problema del requisito 1 (4.1). L'unica cosa che differisce dal caso precedente è il

testing. Infatti in questo caso viene anche testato l'evento **constraint**, definito in precedenza, usato per modificare la base di conoscenza della mind.

```
QActor testanalysis context ctxConsoleAnalysis {
    Plan init normal [
        println("Test ready");
        delay 5000;
        emit constraint : constraint(temp, 20);
        delay 1000;
        forward mindrobotanalysis -m moveRobot : usercmd( consoleGui( startBot ) );
        delay 1000;
        forward mindrobotanalysis -m moveRobot : usercmd( consoleGui( stopBot ) );
        delay 2000;
        emit constraint : constraint(temp, 28);
        delay 2000;
        forward mindrobotanalysis -m moveRobot : usercmd( consoleGui( startBot ) )
    ]
}
```

Figura 18: Test analisi del problema

Il test in un primo caso porta la temperatura a 20 gradi Celsius - ancora valida, e in un secondo caso a 28 gradi Celsius - temperatura non più valida. Nel secondo caso il robot non dovrà muoversi. Questo test ci ha dimostrato che l'interazione e il comportamento della mind funzionano come dovrebbero, quindi passiamo alla progettazione riprendendo le componenti introdotte nel requisito precedente.

5.2 Progettazione

In fase di progettazione ci siamo accorti che la soluzione migliore, per rendere il sistema il più distribuito possibile, è definire un nuovo contesto solo per la **mind**. Questo ci permetterà di eseguire la mind in un qualsiasi nodo indipendente dai nodi del robot virtuale e del robot reale.

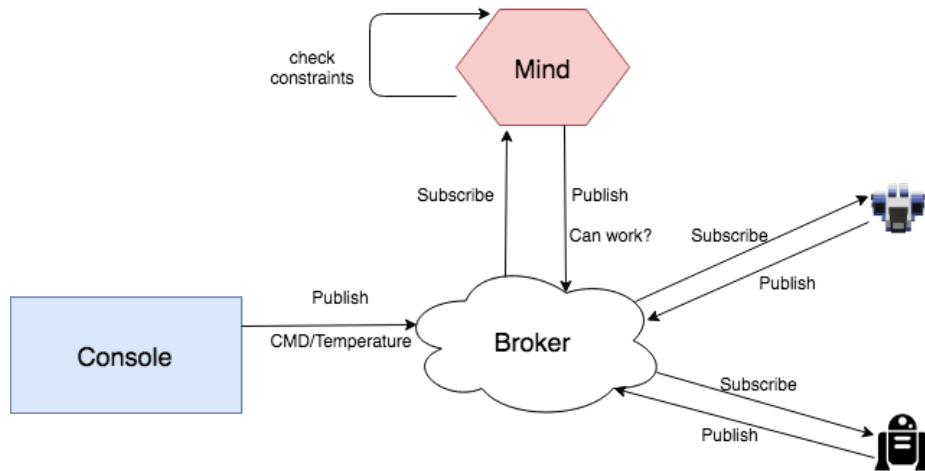


Figura 19: Schema Mind

Importante in questa fase è l'evento constraint che abbiamo introdotto nell'analisi del problema, in quanto permetterà di volta in volta di modificare la temperatura presente nella base di conoscenza della mind.

Rimane però un problema:

- Chi deve aggiornare la temperatura?

Dopo un attento studio e considerando i test effettuati nell'analisi del problema siamo arrivati alla conclusione che deve essere la console ad aggiornarla. In questo modo sulla console potrà essere visualizzata la temperatura relativa alla posizione del robot, in modo tale che anche l'utente autorizzato possa vederla.

Quindi, ripercorrendo quanto detto finora, i passi che saranno eseguiti per l'aggiornamento della temperatura sono i seguenti:

1. La console stabilisce in base alla posizione del robot la temperatura: per rendere le cose più semplici abbiamo deciso che la posizione viene stabilita dall'utente attraverso un apposito campo del suo profilo;
2. Ogni volta che la temperatura cambia viene inviata alla mind: per semplificare abbiamo deciso di aggiornare il valore della temperatura ogni 30 secondi;
3. La mind valuta se la temperatura inviata è al di sotto della soglia indicata.

Rimane ora il problema fondamentale sul come fare tutto ciò per realizzare un secondo prototipo da far visionare al committente; di questo ce ne occuperemo in fase di implementazione.

5.3 Implementazione

Non verranno riportate le implementazioni del **Virtual Robot** e del **Real Robot** in quanto non sono state effettuate modifiche rispetto al requisito precedente.

5.3.1 Implementazione: Mind .

Per realizzare la struttura individuata nella fase di progettazione abbiamo iniziato con l'implementazione del modello della mind. Visto che nell'analisi del problema era stato già definito un modello (Fig. 16) che ne raccogliesse le informazioni essenziali in ambiente locale, abbiamo adattato tale modello ad un ambiente distribuito.

In figura sono mostrati il **gestore degli eventi** e il **gestore dei messaggi**: il primo si occupa di gestire l'evento constraint ricevuto dalla console (17), il secondo i messaggi relativi al movimento del robot.

Come avevamo già anticipato in fase di analisi e di progettazione, i messaggi (**moveRobot**) relativi ai comandi inviati dalla console verranno gestiti dalla

```

Plan waitPlan[ ]
  transition stopAfter 3600000 //1h
    whenEvent constraint -> handleEvent,
    whenMsg moveRobot -> handleMsg
  finally repeatPlan

```

Figura 20: Gestori

mind. Questo ci permetterà di inviare il messaggio ai robot solo se sono soddisfatti i vincoli. Quindi, considerando la base di conoscenza definita in fase di analisi del problema (5.1.1), il gestore dei messaggi diventerà quello mostrato nella figura che segue.

```

Plan handleMsg resumeLastPlan [
  printCurrentMessage;
  onMsg moveRobot : usercmd( robotgui(w(X)) ) ->{
    [ !? checkTemperature(cold) ] {
      publishMsg "unibo/gasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(w(low)) );
      publishMsg "unibo/gasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(w(low)) )
    }
    else   println("Too hot to work")
  };
  onMsg moveRobot : usercmd( robotgui(s(X)) ) ->{
    [ !? checkTemperature(cold) ] {
      publishMsg "unibo/gasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(s(low)) );
      publishMsg "unibo/gasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(s(low)) )
    }
    else   println("Too hot to work")
  };
  onMsg moveRobot : usercmd( robotgui(a(X)) ) ->{
    [ !? checkTemperature(cold) ] {
      publishMsg "unibo/gasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(a(low)) );
      publishMsg "unibo/gasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(a(low)) )
    }
    else   println("Too hot to work")
  };
  onMsg moveRobot : usercmd( robotgui(d(X)) ) ->{
    [ !? checkTemperature(cold) ] {
      publishMsg "unibo/gasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(d(low)) );
      publishMsg "unibo/gasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(d(low)) )
    }
    else   println("Too hot to work")
  };
  onMsg moveRobot : usercmd( robotgui(h(X)) ) -> {
    [ !? checkTemperature(cold) ] {
      publishMsg "unibo/gasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(h(low)) );
      publishMsg "unibo/gasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(h(low)) )
    }
    else   println("Too hot to work")
  }
]

```

Figura 21: Mind: HandleMsg

Nel codice sono state inserite delle **guardie** che verificano se una condizione è soddisfatta o meno.

```
[ !? checkTemperature(cold) ]
```

Se la condizione è soddisfatta, allora la mind pubblica un messaggio su mqtt che indica ai robot il movimento da effettuare, altrimenti stampa la seguente stringa: "Too hot to work".

Dal canto suo il **gestore dell'evento**, ogni volta che riceverà l'evento **constraint**, dovrà **aggiornare** la temperatura presente nella **base di conoscenza** e verificare se i vincoli sono rispettati. Nel caso in cui **non** lo siano deve inviare al robot un messaggio di Stop (sempre attraverso mqtt come nel caso precedente).

```
Plan handleEvent resumeLastPlan []
printCurrentEvent;
onEvent constraint : constraint(temp, V) -> ReplaceRule currentTemperature(X) with currentTemperature(V);
[ !? checkTemperature(hot) ]{
    publishMsg "unibo/qasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(h(low)) );
    publishMsg "unibo/qasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(h(low)) )
}
]
```

Figura 22: Mind: HandleEvent

5.3.2 Implementazione: Aggiornamento Console .

La console, a fronte delle considerazioni fatte nei paragrafi precedenti, subirà delle leggere modifiche. Infatti ora dovrà anche ricavare la temperatura relativa alla posizione del robot; come già detto in fase di progettazione, per la **posizione** del robot ci baseremo su un campo inserito dall'utente durante la registrazione. Per ricavare la temperatura, invece, viene usata un'apposita **API**(Fig. 23).

Come stabilito in fase di progettazione, verrà emesso un evento ogni **30** secondi che conterrà il valore della temperatura relativa alla posizione stabilita dall'utente autorizzato in fase di registrazione.

In Fig. 24 viene riportato il clientRest, ovvero colui che effettuerà la vera e propria chiamata **REST** al link <https://query.yahooapis.com>, per ottenere la temperatura.

```

1  const restClient = require('../utils/restClient');
2  const clientMqtt = require('../utils/mqttUtils');
3
4  const weatherIntervals = {};
5
6  exports.emit_weather_temperature = (user) => {
7      if (user.city) {
8          weather_emit(user.city);
9          if (weatherIntervals[user._id]) {
10              clearInterval(weatherIntervals[user._id]);
11          }
12          weatherIntervals[user._id] = setInterval(() => weather_emit(user.city), 30000); // 30 seconds
13      }
14  };
15
16  exports.clear_weather_emitter = (userId) => {
17      clearInterval(weatherIntervals[userId]);
18  };
19
20  exports.clear_all_weather = () => {
21      Object.keys(weatherIntervals).forEach(userId => clearInterval(weatherIntervals[userId]));
22  };
23
24  const weather_emit = (city) => {
25      const queryString = "https://query.yahooapis.com/v1/public/yql?q=select item from weather.forecast"
26      + "where woeid in (select woeid from geo.places(1) where text='" + city + "') and u='c'&format=json"
27      restClient.get_weather_temperature(queryString)
28          .then(temperature => {
29              console.log("Temperature of " + city + " = " + temperature);
30              let eventstr = 'msg(constraint.event.js,mindrobot,constraint(temp, ' + temperature + '),1)';
31              console.log("emit > " + eventstr);
32              clientMqtt.publish(eventstr);
33          })
34          .catch(err => next(err));
35  };
36

```

Figura 23: API Temperatura

```
1 const RestClient = require('node-rest-client').Client;
2 const client = new RestClient();
3
4 exports.get = (query) => {
5     return new Promise((resolve, reject) => {
6         client.get(query, (data, response) => {
7             if(data){
8                 resolve(data);
9             } else {
10                 reject("No result");
11             }
12         })
13     })
14 };
15
16 exports.get_weather_temperature = (query) => {
17     return new Promise((resolve, reject) => {
18         client.get(query, (data, response) => {
19             if( data.query.results == null ){
20                 reject("No result");
21             } else {
22                 if( data.query.results.channel.item != null ){
23                     resolve(data.query.results.channel.item.condition.temp);
24                 } else {
25                     reject("No item result");
26                 }
27             }
28         })
29     })
30};
```

Figura 24: ClientRest

A questo punto non ci restava altro che mostrare la temperatura direttamente sulla console dell'utente autenticato. Per farlo è bastato aggiornare la componente **Console.js** di React. Il risultato finale a livello grafico è quello mostrato in figura.

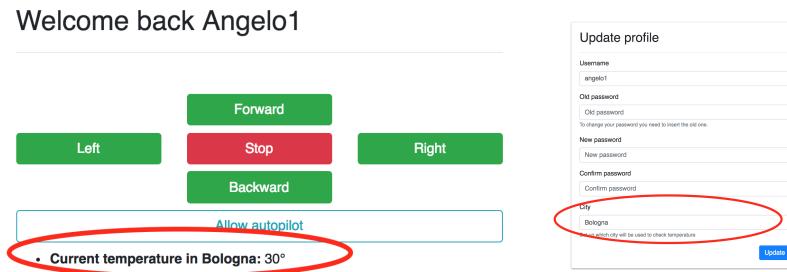


Figura 25: Console aggiornata

Per effettuare altri test, come si può notare dalla figura, è possibile modificare la città anche una volta che l'utente si è registrato direttamente dalla pagina della console.

Degno di nota è il fatto che la pubblicazione dell'evento constraint su mqtt viene fatto di volta in volta dal web server presente nella console.

Arrivati a questo punto possiamo presentare un altro prototipo funzionante al commitente, che soddisfa il secondo requisito.

6 Requisito 3

6.1 Analisi del problema

Dall'analisi dei requisiti si deduce che occorre introdurre un nuovo vincolo: *se l'orario corrente non è compreso all'interno di un intervallo (ad esempio non è tra le 7 e le 10 di mattina) il robot deve fermare la sua attività.* Da qui si capisce bene come il fatto di aver introdotto un componente (**mind**) che si occupi del controllo del rispetto dei vincoli sia la scelta migliore.

Dopo l'interazione con l'utente in fase di analisi dei requisiti si è stabilito che l'orario deve essere **newtoniano** e relativo al **fuso orario** del **robot**. Quindi la scelta più conveniente sarà quella di far stabilire l'orario del robot alla mind e di volta in volta inviarlo alla console, in modo tale che anche l'utente autorizzato possa visionarlo.

6.1.1 Analisi del problema: Mind .

Alla nostra base di conoscenza andiamo ad introdurre una nuova regola che controlli che il nuovo vincolo sia rispettato. Bisogna anche tenere in considerazione che, allo stesso tempo, anche il vincolo della temperatura deve essere soddisfatto. Quindi partendo dalla base di conoscenza introdotta in precedenza (5.1.1), introduciamo le seguenti regole:

```
startTime(7).  
endTime(10).  
...
```

Figura 26: Intervallo di tempo (anche se non riportata in figura ci sarà una regola **currentTime** che indica l'ora corrente)

Tali regole ci permettono di stabilire l'intervallo di tempo in cui il robot può svolgere la sua attività. Per controllare, invece, che questo intervallo venga rispettato bisogna introdurre un'ulteriore regola, mostrata nella figura 27.

```

checkTime(X):-
    startTime(START),
    endTime(END),
    currentTime(CURRENT),
    eval(get, CURRENT, START),
    eval(let, CURRENT, END).

```

Figura 27: Controllo del vincolo

L'unica cosa che rimane da fare prima di testare se la base di conoscenza funziona correttamente è controllare che anche il vincolo della temperatura sia soddisfatto. Per farlo viene introdotta un'ultima regola illustrata nella figura 28.

```

checkConstraints(X):-
    checkTemperature(cold),
    checkTime(X).

```

Figura 28: Controllo dei vincoli

Per testare che la base di conoscenza funzioni, utilizziamo l'evento definito in precedenza(Fig. 17) per aggiornare l'orario ad un orario non valido.

Il test (Fig. 30) ha avuto esito positivo, quindi siamo certi che la base di conoscenza abbia il comportamento aspettato.

Dopo aver stabilito **come** effettuare i controlli, usando il modello della mind definito nel requisito 2 con la base di conoscenza appena ridefinita, vedremo in fase di progettazione come mettere tutto insieme.

```

QActor testanalysistime context ctxConsoleAnalysis {
    Plan init normal[  

        println("Test time ready")
    ]  

    switchTo waitPlan  

    Plan waitPlan[  

        transition stopAfter 3600000 //1h  

        whenEvent constraint -> handleEvent  

        finally repeatPlan
    ]  

    Plan handleEvent resumeLastPlan [  

        onEvent constraint : constraint(tempo, V) -> printCurrentEvent
    ]
}

```

Figura 29: Test del requisito 3

6.2 Progettazione

In fase di progettazione andremo a stabilire come ottenere l'orario corrente e come comunicarlo al sistema. Per ottenere l'orario abbiamo deciso di utilizzare un metodo statico che vada a ricavare l'ora corrente e che aggiorni la base di conoscenza della **mind**. Una volta che la mind avrà usato questo metodo, andrà a comunicare il risultato ottenuto alla console, attraverso il modello publish/subscribe che caratterizza il nostro sistema.

Il metodo statico che andremo a definire in fase di implementazione dovrà essere richiamato ogni volta che la console invia un comando da eseguire, in modo tale che la mind, prima di verificare che l'orario rispetti il vincolo, possa averlo sempre aggiornato.

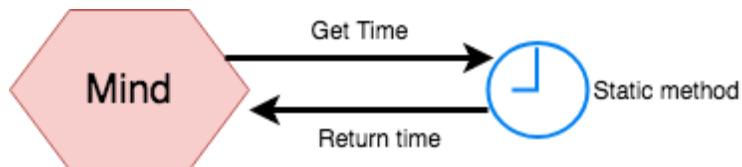


Figura 30: Schema richiesta dell'orario

6.3 Implementazione

6.3.1 Implementazione: Metodo Statico .

Il metodo statico dovrà ricavare l'orario corrente e per farlo userà la classe `Date` di Java , in particolare per avere il formato adatto dell'ora:

```
SimpleDateFormat sdf = new SimpleDateFormat("HH");
String hours = sdf.format(new Date());
```

Una volta ottenuto l'orario, dovrà aggiornare la base di conoscenza della **mind**.

Per poter modificare la base di conoscenza dovrà avere accesso ai metodi del ***QActor*** .

Una volta che si ha il ***QActor*** si potrà accedere alla base di conoscenza e aggiornare l'orario corrente:

```
myActor.replaceRule("currentTime(X)", "currentTime("+hours+"));
```

La definizione del metodo sarà quindi la seguente:

```
//myActor corrisponde al \qa che ha invocato il metodo
public static void getHours(QActor myActor)
```

6.3.2 Implementazione: Mind .

La Mind, arrivati a questo punto dell'implementazione, dovrà aggiornare l'ora ogni volta che riceve un comando dalla console. Sarà quindi necessario apportare una modifica al gestore dei messaggi (Fig. 21). Da notare che per ogni richiesta

```

Plan handleMsg resumeLastPlan [
    javaRun it.unibo.utils.customDate.getHours();
    /* Pubblico un messaggio relativo al tempo*/
    demo currentTime(V); //recupero il tempo (Attualmente è 8 per provare che funzioni)
    [ ?? goalResult(currentTime(R)) ] publishEvent "unibo/gasys" -e constraint : constraint(tempo, R);

    printCurrentMessage;
    onMsg moveRobot : usercmd( robotgui(w(X)) ) ->{
        [ !? checkConstraints(X) ] {
            publishMsg "unibo/gasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(w(low)) );
            publishMsg "unibo/gasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(w(low)) )
        }
        else
            println("Too hot to work or out of time")
    };
    onMsg moveRobot : usercmd( robotgui(s(X)) ) ->{
        [ !? checkConstraints(X) ] {
            publishMsg "unibo/gasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(s(low)) );
            publishMsg "unibo/gasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(s(low)) )
        }
        else
            println("Too hot to work or out of time")
    };
    onMsg moveRobot : usercmd( robotgui(a(X)) ) ->{
        [ !? checkConstraints(X) ]{
            publishMsg "unibo/gasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(a(low)) );
            publishMsg "unibo/gasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(a(low)) )
        }
        else
            println("Too hot to work or out of time")
    };
    onMsg moveRobot : usercmd( robotgui(d(X)) ) ->{
        [ !? checkConstraints(X) ]{
            publishMsg "unibo/gasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(d(low)) );
            publishMsg "unibo/gasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(d(low)) )
        }
        else
            println("Too hot to work or out of time")
    };
    onMsg moveRobot : usercmd( robotgui(h(X)) ) -> {
        [ !? checkConstraints(X) ] {
            publishMsg "unibo/gasys" for "virtualrobotexecutor" -m execMoveRobot : usercmd( robotgui(h(low)) );
            publishMsg "unibo/gasys" for "realrobotexecutor" -m execMoveRobot : usercmd( robotgui(h(low)) )
        }
        else
            println("Too hot to work or out of time")
    }
]

```

Figura 31: Gestore messaggi

di movimento da parte della console si va a controllare che il predicato **check-Constraints** (definito nella Fig. 28) restituisca true e solo in quel caso si inoltra il messaggio ai robot. Inoltre, la mind dovrà aggiornare il sistema notificando l'ora appena ricevuta.

```
[ ?? goalResult(currentTime(R)) ]  
publishEvent "unibo/qasys" -e constraint : constraint(tempo, R);
```

7 Requisito 4

7.1 Analisi del problema

La scelta fatta dopo aver analizzato questo requisito è stata quella di dividerlo in due parti: una prima parte che si concentra principalmente sul led e una seconda parte incentrata sugli ostacoli da evitare presenti nella stanza.

In questo capitolo ci occuperemo soltanto della prima parte, senza fare distinzione, in fase di analisi, tra il **led** e il **Led Hue Lamp** presente all'interno della casa.

Anche in questo caso la **mind** avrà un ruolo fondamentale, in quanto dovrà segnalare al led se il robot è in movimento o è fermo, in modo tale che possa stabilire se lampeggiare o meno.

Prima di vedere come realizzare l'interazione tra il led e la mind, cerchiamo di realizzare un modello del led che catturi gli aspetti essenziali del suo comportamento.

7.1.1 Analisi del problema: Led Prima Versione In questa prima versione del led, per la realizzazione del modello, useremo il linguaggio custom *QActor*, in quanto con questo potremo realizzare un attore che lavora in un contesto differente e quindi simulare il fatto che il led possa trovarsi in un nodo indipendente da quello della mind. In particolare l'attore rappresentante il led effettuerà una stampa della stringa: "*Led blink*" se il robot è in movimento e "*Led not blink*" se il robot è fermo (Fig. 32) .

L'evento `Event ctrlEvent : ctrlEvent(CMD)` verrà emesso dalla mind per segnalare che il robot è in movimento (Fig. 33).

A questo punto possiamo effettuare il test sull'interazione tra mind e led. Per farlo, come già fatto nei casi precedenti, basta simulare l'invio dei comandi da parte di una console (Fig. 34).

```

| QActor ledanalysis context ctxLedAnalysis {
|   Plan init normal [
|     println("Led ready!");
|     println("Led not blink")
|   ]
|   switchTo waitMove
|
|   Plan waitMove[ ]
|     transition stopAfter 3600000 //1h
|       whenEvent ctrlEvent -> handleEvent
|       finally repeatPlan
|
|   Plan handleEvent resumeLastPlan [
|     printCurrentEvent;
|     onEvent ctrlEvent : ctrlEvent(on) -> println("Led blink");
|     onEvent ctrlEvent : ctrlEvent(off) -> println("Led not blink")
|   ]
}

```

Figura 32: Modello Led

```

Plan waitPlan[ ]
transition stopAfter 3600000 //1h
whenEvent constraint -> handleEvent,
whenMsg moveRobot -> handleMsg
finally repeatPlan
|
Plan handleEvent resumeLastPlan [
  printCurrentEvent;
  onEvent constraint : constraint(temp, V) -> ReplaceRule currentTemperature(X) with currentTemperature(V);
  [ !? checkTemperature(hot) ] {
    forward robotexecutoranalysis -m execMoveRobot : usercmd( consoleGui( stopBot ) );
    delay 100;
    emit ctrlEvent : ctrlEvent(off)
  }
]
|
Plan handleMsg resumeLastPlan [
  printCurrentMessage;
  onMsg moveRobot : usercmd( consoleGui( startBot ) ) ->{
    [ !? checkConstraints(X) ]{
      forward robotexecutoranalysis -m execMoveRobot : usercmd( consoleGui( startBot ) );
      delay 100;
      emit ctrlEvent : ctrlEvent(on)
    }
    else println("Too hot to work")
  };
  onMsg moveRobot : usercmd( consoleGui( stopBot ) ) ->{
    [ !? checkConstraints(X) ]{
      forward robotexecutoranalysis -m execMoveRobot : usercmd( consoleGui( stopBot ) );
      delay 100;
      emit ctrlEvent : ctrlEvent(off)
    }
    else println("Too hot to work")
  }
]

```

Figura 33: Analisi del problema: mind

```

QActor testanalysis context ctxConsoleAnalysis {
  Plan init normal [
    println("Test ready");
    delay 5000;
    forward mindrobotanalysis -m moveRobot : usercmd( consoleGui( startBot ) );
    delay 1000;
    forward mindrobotanalysis -m moveRobot : usercmd( consoleGui( stopBot ) );
    delay 4000;
    forward mindrobotanalysis -m moveRobot : usercmd( consoleGui( startBot ) );
    delay 2000;
    forward mindrobotanalysis -m moveRobot : usercmd( consoleGui( stopBot ) )
  ]
}

```

Figura 34: Test interazione

7.1.2 Analisi del problema: Led Seconda Versione .

Per simulare il comportamento del led useremo un oggetto Mock (realizzato in Java) fornитoci dalla software house. In particolare:

- se il led è spento l'oggetto Mock avvierà una piccola Gui di colore rosso;
- se il led è acceso l'oggetto Mock avvierà una Gui più grande rispetto alla precedente.

L'oggetto Mock inoltre espone un metodo che permetterà di avere un alternarsi tra interfaccia grande e interfaccia piccola, in pratica ci permette di simulare il "lampeggiare" del nostro led.

Il modello del led, dopo l'introduzione dell'oggetto Mock, sarà quello mostrato in figura 35.

```
> QActor ledmockledanalysis context ctxLedMockLedAnalysis {
>   Plan init_normal [
>     println( "ledmock starts" );
>     javaRun it.unibo.ledmockgui.customGui.createCustomLedGui();
>     delay 1000;
>     javaRun it.unibo.ledmockgui.customGui.setLedBlink("on");
>     delay 6000;
>     javaRun it.unibo.ledmockgui.customGui.setLedBlink("off");
>     println("Siamo qui!")
>   ]
>   switchTo waitMove
>
>   Plan waitMove[ ]
>   transition stopAfter 3600000 //1h
>     whenEvent ctrlEvent -> handleEvent
>     finally repeatPlan
>
>   Plan handleEvent_resumeLastPlan [
>     printCurrentEvent;
>     onEvent ctrlEvent : ctrlEvent(on) -> javaRun it.unibo.ledmockgui.customGui.setLedBlink("on");
>     onEvent ctrlEvent : ctrlEvent(off) -> javaRun it.unibo.ledmockgui.customGui.setLedBlink("off")
>   ]
> }
```

Figura 35: Modello Led con l'utilizzo dell'oggetto Mock

Il test è identico a quello precedente; la differenza sta nel fatto che mentre prima avevamo una semplice stampa ora abbiamo una vera e propria simulazione del comportamento del led.

7.2 Progettazione

In fase di progettazione andiamo a sostituire il *QActor* aggiunto in fase di analisi con un ulteriore componente per separare le parti già esistenti da quella che si occupa della gestione del led, in quanto, d'ora in avanti, si avrà una distinzione tra il **led fisico** sul real robot e il **led hue lamp** sul virtual robot.

Questo nuovo componente verrà realizzato cercando di ottenere una struttura generale ed indipendente, utilizzabile per entrambe le tipologie di led e facilmente estendibile in futuro.

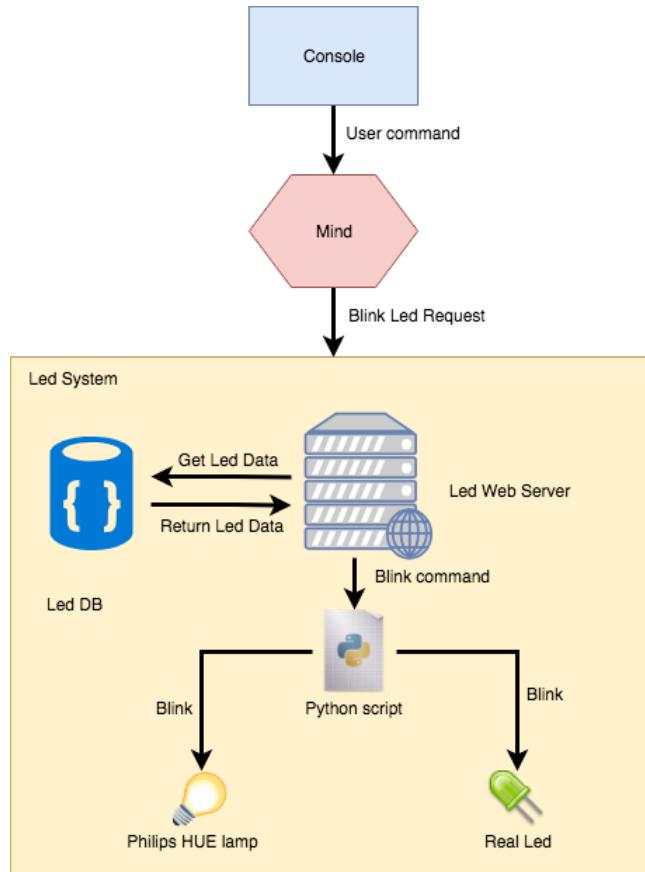


Figura 36: Sistema led

Come si può notare dalla figura (36) questo sistema si compone principalmente di:

- un **web server** (scritto in NodeJS) che avrà il compito di comunicare tramite chiamate HTTP con la mind;
- Di un **database** (MongoDB) in cui saranno salvate le informazioni relative ai led presenti nel sistema;
- Di uno **script python** in grado di comunicare tramite socket con il web server e che avrà il compito di far accendere i led.

7.3 Implementazione

7.3.1 Implementazione: Client Rest .

L'implementazione di questo requisito ha richiesto una modifica nella mind; infatti è stato eliminato l'evento introdotto in fase di analisi del problema (`Event ctrlEvent : ctrlEvent(CMD)`). Al suo posto è stata inserita una chiamata HTTP realizzata mediante un metodo statico custom creato appositamente per questa funzione (il **Client Rest**).

Questo metodo prende in ingresso 3 parametri:

- Il comando da inviare al web server dei led (**true** per far blinkare il led e **false** per farlo smettere);
- L'eventuale colore del led nel caso del Led Hue;
- L'uri da utilizzare per effettuare la chiamata HTTP.

```

public static void sendPutBlink(QActor qa, String value, String color, String url) {
    try {
        URL urlblink = new URL(url);
        System.out.println(urlblink);

        HttpURLConnection connection = (HttpURLConnection) urlblink.openConnection();
        connection.setConnectTimeout(5000); //5 secs
        connection.setReadTimeout(5000); //5 secs

        connection.setRequestMethod("PUT");
        connection.setDoOutput(true);
        connection.setRequestProperty("Content-Type", "application/json");

        JSONObject json = new JSONObject();
        for (String key : data.keySet()) {
            json.put(key, data.get(key));
        }
        json.put("value", value);
        json.put("color", color);
        System.out.println("Json object to send: " + json.toString());
        OutputStreamWriter out = new OutputStreamWriter(connection.getOutputStream());
        out.write(
                json.toString());
        out.flush();
        out.close();

        int res = connection.getResponseCode();

        System.out.println(res);

        InputStream is = connection.getInputStream();
        BufferedReader br = new BufferedReader(new InputStreamReader(is));
        String line = null;
        System.out.println("Response received: ");
        while((line = br.readLine() ) != null) {
            System.out.println(line);
        }
        connection.disconnect();
    } catch (IOException | JSONException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}

```

Figura 37: Codice del metodo statico per la chiamata HTTP

Il metodo, come si evince dalla figura (37), prepara una chiamata PUT al web server dei led, crea un JSON Object in cui va ad inserire il body della chiamata (ovvero due parametri value e color) ed infine invia la richiesta HTTP e stampa la risposta ricevuta dal server.

7.3.2 Implementazione: Led System .

Nell'implementazione di questo componente ci siamo concentrati inizialmente nella realizzazione del web server e dello script python. Infine sono stati creati all'interno dello script python degli adapter specifici per le due tipologie di led necessarie.

7.3.3 Implementazione: Web Server .

Il Web Server, ogni volta che riceverà una richiesta dalla mind, dovrà inviare sulla socket stabilita con lo script python il comando richiesto. In tale richiesta, oltre alla specifica del **comando**, possono essere presenti:

- il **codice univoco** della lampadina sulla quale attuare il comando (**obbligatorio**);
- il **colore** della lampadina (**opzionale** in quanto valido solo per la Led Hue Lamp);

Una volta arrivata la richiesta, il web server verificherà che il codice esista all'interno del suo database. Successivamente il comando verrà interpretato in modo tale da rispettare il requisito. Se il comando corrisponde a **true** il web server invia sulla socket, seguendo un intervallo, un messaggio di accensione e uno di spegnimento, permettendo al led di lampeggiare; se il comando corrisponde a **false** il web server "distruggerà" l'intervallo e invierà sulla socket un messaggio di spegnimento.

```

const blink_lamp = (req, res, next) => {
  const codeLamp = req.params.code;
  let lampPromise = Lamp.findOne({code: codeLamp})
    .then(lamp => {
      if (lamp) {
        const value = req.body.value;
        lamp.value = value;
        return lamp.save().then(lamp => lamp).catch(err => next(err));
      } else {
        res.header('Content-Type', 'application/json');
        res.status(400);
        res.json({
          error: 'Lamp id doesn\'t exist'
        });
      }
    });
  lampPromise.then(lamp => {
    socketServer.emitAll('value', lamp.value);
    if(lamp.value){
      if(refreshIntervalId == 0){
        /*Blinking (se già sta blinkando non viene chiamata)*/
        refreshIntervalId = setInterval(()=>{
          socketServer.emitAll('value', lamp.value);
          lamp.value = !lamp.value;
        }, 2000);
      }
    }else{
      if(refreshIntervalId != 0){
        clearInterval(refreshIntervalId);
        refreshIntervalId = 0;
      }
      socketServer.emitAll('value', lamp.value)
    }
    res.header('Content-Type', 'application/json');
    res.status(200);
    res.json({
      lamp: lamp
    });
  })
);
}

```

Figura 38: Codice della gestione della richiesta (web server)

7.3.4 Implementazione: Script Python .

Al lancio di questo script dovrà essere specificato se il led, sul quale si vuole compiere un'azione (accensione o spegnimento), è quello posto sulla raspberry o quello posto all'interno della casa (Led Hue Lamp). Proprio per questo motivo nei paragrafi successivi verranno introdotti due script che descrivano queste azioni.

Una volta lanciato, lo script aprirà una socket con il web server e si metterà in attesa di un comando. Ogni volta che sarà presente un valore sulla socket verrà richiamata una **callback**, a seconda del led specificato, che effettuerà l'operazione richiesta. Questa parte verrà trattata nel dettaglio nei paragrafi successivi.

```
def main(argv):
    arguments_result_error = reed_arguments(argv)
    if arguments_result_error is True:
        print_help()
    else:
        if start_node:
            print("Starting node will cause problem on stop application, need to force to quit")
            node_thread = Thread(target=start_node_action)
            node_thread.start()

    print("Waiting to connect to: " + host + " port: " + str(port))
    client_socket = ClientSocketIO(host, port)
    client_socket.connect()

    # client_socket.emit('Ciao node')
    socket_thread = Thread(target=client_socket.wait)
    socket_thread.start()

    if led_gpio:
        print("LED GPIO", end='\n\n')
        client_socket.add_new_event('value', gpio_on_value_response)
    else:
        print("LAMP VIRTUAL", end='\n\n')
        lamp = Lamp('Philips hue lamp', client_socket)
        lamp.mainloop()

if __name__ == '__main__':
    main(sys.argv)
```

Figura 39: Main Script python

7.3.5 Implementazione: Led .

In questa implementazione, realizzata anch'essa in **python**, sono presenti due funzioni: una per accendere e una per spegnere il led. Entrambe verrannoificate dalla funzione di callback, impostata nello script python iniziale, che, una volta ricevuto il comando, stabilirà quale sarà la funzione da invocare.

```
1 import RPi.GPIO as GPIO
2
3 LED = 8
4 GPIO.setmode(GPIO.BCM)
5 GPIO.setwarnings(False)
6 GPIO.setup(LED, GPIO.OUT)
7
8
9 def led_on():
10     print("Led is ON")
11     GPIO.output(LED,True)
12
13
14 def led_off():
15     print("Led is OFF")
16     GPIO.output(LED,False)
17
18
19 def on_value_response(value):
20     if value == True:
21         led_on()
22     elif value == False:
23         led_off()
24
```

Figura 40: Led Robot Reale

Per accedere ai PIN della raspberry in python esiste la libreria **GPIO** che, una volta stabilita la modalità di accesso (nel nostro caso **BCM** Broadcom SOC channel), permette di stabilire se accedervi in INPUT o in OUTPUT. Nel nostro caso il PIN dove il led è posto è il numero 8; questo sarà quindi impostato in OUTPUT.

Una volta effettuate le operazioni sopra elencate, le funzioni useranno la libreria GPIO per impostare il valore del PIN.

7.3.6 Implementazione: Simulazione Led Hue Lamp .

L'implementazione del **Led Hue Lamp** richiesto dal committente ha necessitato la creazione di un componente mock che simulasse il comportamento di tale lampadina, in quanto quest'ultima non è attualmente disponibile nella nostra software house.

La Led Hue Lamp mock è quindi stata realizzata in **python**, in quanto era già stato previsto un ambiente python in grado di comunicare con il web server per l'accensione del led fisico tramite i pin GPIO; quindi è stata realizzata una classe che utilizza il package grafico di python (**tkinter**) per creare una finestra di piccole dimensioni (70px) che simuli una lampadina spenta. Questa classe prevede poi un metodo da invocare quando si vuole accendere la lampada che va ad aumentare la dimensione della finestra (a 500px, lampadina accesa).

Infine in questa classe nel costruttore viene impostata la callback, invocata al presentarsi di un valore sulla socket definita nello script python(7.3.4), che permetterà di modificare la dimensione della finestra.

```

class Lamp:
    def __init__(self, title, client_socket):
        self.client_socket = client_socket
        self.root = Tk()
        self.root.title(title)
        self.root.geometry("+400+250")
        self.root.configure({
            "background": "#ff0000",
            "width": OFF_SIZE['width'],
            "height": OFF_SIZE['height']
        })
        self.root.protocol("WM_DELETE_WINDOW", self.on_closing)
        # socket event handlers
        self.client_socket.add_new_event('value', self.on_value_response)
        self.client_socket.add_new_event('color', self.on_color_response)

    def on_value_response(self, value):
        print("Value lamp = ", value)
        if value == True:
            self.change_size(dimensions=ON_SIZE)
        elif value == False:
            self.change_size(dimensions=OFF_SIZE)

    def on_color_response(self, color):
        print("Color lamp = ", color)
        self.root.configure({
            "background": color
        })

    def change_size(self, event=None, dimensions=OFF_SIZE):
        print(event)
        self.root.configure({
            "width": dimensions['width'],
            "height": dimensions['height']
        })

    def on_closing(self):
        print("Closing lamp")
        self.client_socket.emit_close()
        self.client_socket.close()
        self.root.quit()
        exit(0)

    def mainloop(self):
        self.root.mainloop()

```

Figura 41: Classe della lampadina HUE

7.4 Progettazione: Refactoring

Dopo aver realizzato un ulteriore prototipo funzionante per il committente, ci siamo resi conto che una soluzione migliore sarebbe stata quella di estrarre la base di conoscenza dalla mind. In questo modo la mind avrebbe avuto un carico applicativo ridotto, ma sarebbe comunque riuscita ad accedere a tutte le informazioni essenziali di tutti i componenti; un ulteriore vantaggio sarebbe stato quello di poter separare la base di conoscenza dalla logica applicativa.

La base di conoscenza verrà quindi inserita all'interno di un file Prolog chiamato **Resource model** (modello delle risorse). La struttura della mind diventerà quella illustrata in figura 42.

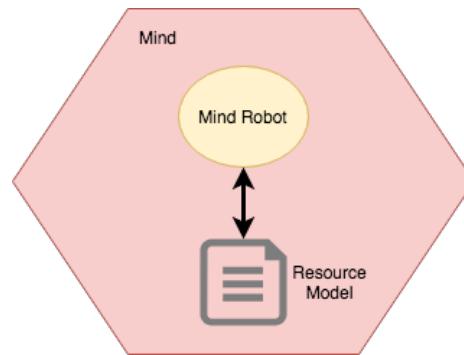


Figura 42: Mind con resource model

7.5 Implementazione: Refactoring

La mind, separata dalla base di conoscenza (le Rules), al suo avvio dovrà caricare nel sistema il file Prolog definito **Resource model** in fase di progettazione (Fig. 43).

Il resource model si compone principalmente di 3 parti:

```

Plan init normal [
    demo consult("./resourceModel.pl");
    println("Mind robot ready")
]

```

Figura 43: Mind inizializza il resource model

- La prima contiene lo stato dei componenti (o risorse) del sistema **Model** e un predicato da usare per leggere questi valori (Fig. 44);
- La seconda prevede un predicato (**changeModelItem**) da utilizzare ogni volta che si vuole modificare uno dei modelli presenti nella prima parte (Fig. 45);
- La terza contiene le azioni da eseguire quando uno delle proprietà del model viene modificata. Queste azioni verranno usate per verificare la correttezza dei vincoli (inseriti nei requisiti precedenti) e per notificare il sistema degli eventuali cambiamenti (tramite l'emissione di eventi).

```

model( type(executor, virtualRobot), name(soffritti), value(true) ).  

model( type(executor, realRobot), name(uffolo), value(true) ).  

model( type(actuator, leds), name(1), value(off) ).  

model( type(actuator, leds), name(2), value(off) ).  

model( type(sensor, temperature), name(cityTemperature), value(12) ).  

model( type(sensor, clock), name(clock1), value(8) ).  

model( type(sensor, sonarVirtual), name(sonar1), value(0)).  

model( type(sensor, sonarVirtual), name(sonar2), value(0)).  

model( type(sensor, sonarRobot), name(sonarVirtual), value(0)).  

model( type(sensor, sonarRobot), name(sonarReal), value(0)).

getModelItem( TYPE, CATEG, NAME, VALUE ) :-  

    model( type(TYPE, CATEG), name(NAME), value(VALUE) ).
```

Figura 44: Modelli delle componenti

```

changeModelItem( CATEG, NAME, VALUE ) :-  

    replaceRule(  

        model( type(TYPE, CATEG), name(NAME), value(_) ),  

        model( type(TYPE, CATEG), name(NAME), value(VALUE) )  

    ),!  

    ( changedModelAction(CATEG, NAME, VALUE)  

    ; true ).
```

Figura 45: Change Model Item

```

changedModelAction( clock, clock1, V):-
    startTime( START ),
    endTime( END ),
    eval( get, V, START),
    eval( let, V, END),
    changeModelItem( virtualRobot, soffritti, true),
    changeModelItem( realRobot, fuffolo, true),
    getModelItem( sensor, temperature, TEMPNAME, TEMP ),
    changedModelAction( temperature, TEMPNAME, TEMP),
    emitevent( resourceChange, resourceChange( sensor, clock, clock1, on ) ).

changedModelAction( clock, clock1, V):-
    changeModelItem( virtualRobot, soffritti, false),
    changeModelItem( realRobot, fuffolo, false),
    emitevent( resourceChange, resourceChange( sensor, clock, clock1, off ) ).
```

Figura 46: Esempio Change Model Action

Introducendo questa separazione, i controlli, precedentemente definiti nella base di conoscenza, non saranno invocati direttamente dalla mind, ma soltanto al verificarsi di un cambiamento all'interno del Resource model. Al cambiamento di una risorsa (`changeModelItem -> changedModelAction`) verrà scatenato un evento che segnalera alla mind il cambiamento effettuato.

7.5.1 Implementazione: Refactoring console .

Il refactoring apportato nel sistema PER introdurre il Resource model ha causato la modifica della console, in quanto per rendere il prodotto più appetibile sul mercato è necessario prevedere la duplicazione delle informazioni relative alle risorse presenti nella Mind anche nel Web Server della console per mostrarle nella GUI dell'utente. Queste informazioni verranno duplicate per garantire il funzionamento dei due componenti (Console e Mind) in maniera indipendente l'uno dall'altro.

Per implementare questa duplicazione la Mind, ogni volta che ci sarà una modifica del suo modello interno, notificherà il Web Server node emettendo su MQTT un evento (`resourceChangeEvent`) il cui payload conterrà le informazioni da modificare. Il Web Server rileverà questi eventi e provvederà ad aggiornare il

proprio Resource model, salvando queste nuove informazione in opportune tabelle del suo DataBase. Infine il Web Server notificherà la GUI React della modifica delle risorse, in modo tale che l'interfaccia possa ricaricare le informazioni aggiornate e mostrarle all'utente (Fig. 47).

System informations			
Name: Robot cleaner			
System sensors			
Temperature of city:	Sydney	Robot time	
Value:	10	Value:	8
Unit of measure:	gradi celsius	Unit of measure:	hour
System executors			
Virtual robot		Real robot	
Name:	Soffritti	Name:	Fuffolo
State:	enabled	State:	enabled
Last action requested:	Forward	Last action requested:	Forward
Robot actuators:		Robot actuators:	
Philips hue lamp mock		Real led on robot	
Status:	Blinking	Status:	Blinking
Robot sensors:		Robot sensors:	
Room sonar 1		Robot real sonar	
Value:	-4	Value:	0
Unit of measure:	point	Unit of measure:	point
Room sonar 2		Possible actions:	
Value:	0	Command	Effect
Unit of measure:	point	w	Forward
Robot virtual sonar	wallDown	d	Right
Value:	point	a	Left
Unit of measure:		s	Backword
Possible actions:		h	Stop
Command	Effect	auto	Autopilot
w	Forward		
d	Right		
a	Left		
s	Backword		
h	Stop		
auto	Autopilot		

Figura 47: Stampa sulla GUI del Resource model

8 Requisito 5

Durante l'analisi del problema del precedente requisito abbiamo rimandato ad fase successiva l'analisi e l'implementazione della funzionalità che permette ai robot di evitare gli ostacoli. Questo perchè abbiamo pensato di integrare quest'ultima con quella relativa all'ultimo requisito, ovvero la costruzione della mappa di una stanza in cui possono essere presenti degli ostacoli.

8.1 Analisi del problema: mappa

Per poter realizzare la mappa durante il processo di pulizia della stanza il robot deve riuscire in qualche modo a tener traccia dei movimenti che effettua.

Abbiamo deciso di procedere dividendo la mappa della stanza in celle che caratterizzano una matrice, dove ogni cella corrisponde alla dimensione del robot.

Per fare ciò occorre tenere in considerazione i seguenti fattori:

- La posizione iniziale del robot;
- La dimensione di una cella;
- Disegnare la mappa come un insieme di celle (così da capire se il robot ha coperto tutta la stanza);
- La posizione iniziale deve essere un angolo della stanza;
- Sui due lati iniziali della stanza non ci devono essere ostacoli.

La **posizione iniziale** del robot è rappresentata dalla posizione che occupa nella stanza al momento dell'inizio della pulizia ed è equivalente alla posizione del sonar 1.

La **dimensione della cella**, rappresentata dalla dimensione del robot, ci aiuterà a capire quali sono gli step che il robot dovrà eseguire. La **dimensione del robot**, come illustrato nella figura 48 , è rappresentata da un valore R che - una volta definito - ci consentirà di stabilire per quanti millisecondi il robot deve muoversi ad una certa velocità costante v per effettuare uno step, ovvero l'attraversamento di una cella. Per il calcolo di questo valore R ci siamo

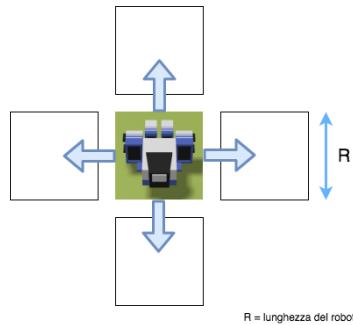


Figura 48: Dimensione della cella

spostati direttamente sull'applicazione del virtual robot, usando il file javascript clientTest messo a disposizione dalla nostra software house. Dalle prove risulta che il valore corretto è di **200ms**, questo significa che gli step che il robot dovrà effettuare sono di 200ms. Ovviamente le stesse prove sono state effettuate anche sul real robot.

Per andare a disegnare la mappa della stanza useremo il numero di celle occupate dal robot fino al raggiungimento del primo ostacolo e il segnale ricevuto dal secondo sonar. Una volta raggiunta questa posizione il robot procederà verso il secondo sonar contando il numero di passi effettuati, così da ottenere la dimensione effettiva della stanza.

La nostra software house ci fornisce anche un **planner** che, data la dimensione della stanza, definirà il piano che il robot dovrà eseguire per pulire la stanzanella sua interezza.

Il **planner** è un agente software che tiene traccia del percorso già effettuato dal robot e che è in grado di restituire la mossa successiva da effettuare in base alla mappa corrente.

8.1.1 Analisi del problema: Ostacoli .

Durante la creazione della mappa - e quindi durante la pulizia della stanza - il robot deve essere in grado di evitare gli ostacoli, come specificato nel requisito 4

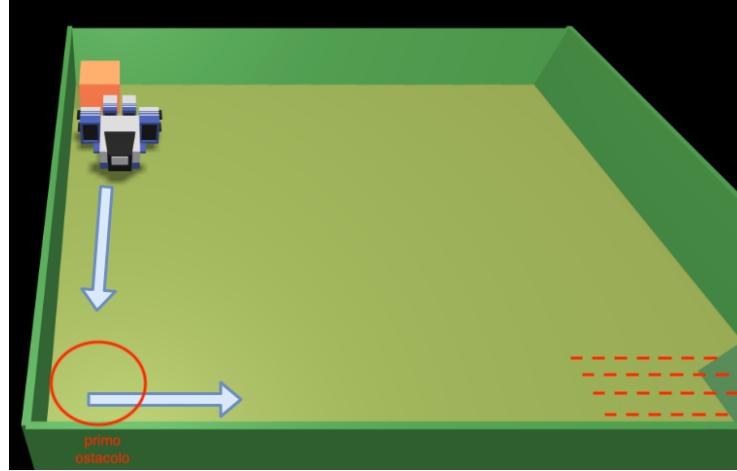


Figura 49: Posizione iniziale del robot

(paragrafo 3.4), facendo distinzione tra gli ostacoli mobili e quelli fissi in quanto solo quest'ultimi devono essere segnati anche nella mappa.

Già in questa fase si individua la nostra strategia da utilizzare per distinguere il tipo di ostacolo. Questa consiste nel tentare, un numero prefissato di volte, di andare avanti come se non ci fossero impedimenti. Se durante questi tentativi riusciremo a superare l'ostacolo, vorrà dire che quest'ultimo è mobile e quindi ignorabile; se invece al termine di questi tentativi l'ostacolo è ancora rilevato, possiamo assumere, con grande probabilità, che quello sia un ostacolo fisso e che quindi andrà segnato anche sulla mappa.

8.2 Progettazione

Da quanto dedotto dall'analisi del problema ci servirà qualcosa che vada ad effettuare le operazioni iniziali, ovvero portare il robot dalla posizione iniziale al secondo sonar seguendo i passi indicati in precedenza. Da qui si capisce la necessità delle ultime due considerazioni iniziali fatte in analisi del problema: la posizione iniziale deve essere un angolo della stanza e sui due lati iniziali della stanza non ci devono essere ostacoli.

Una volta che il robot avrà raggiunto il secondo sonar (end-point) ed avrà calco-

lato la dimensione della mappa, il metodo che si occuperà della sua costruzione dovrà far partire il planner fornитoci dalla nostra software house. L'architettura

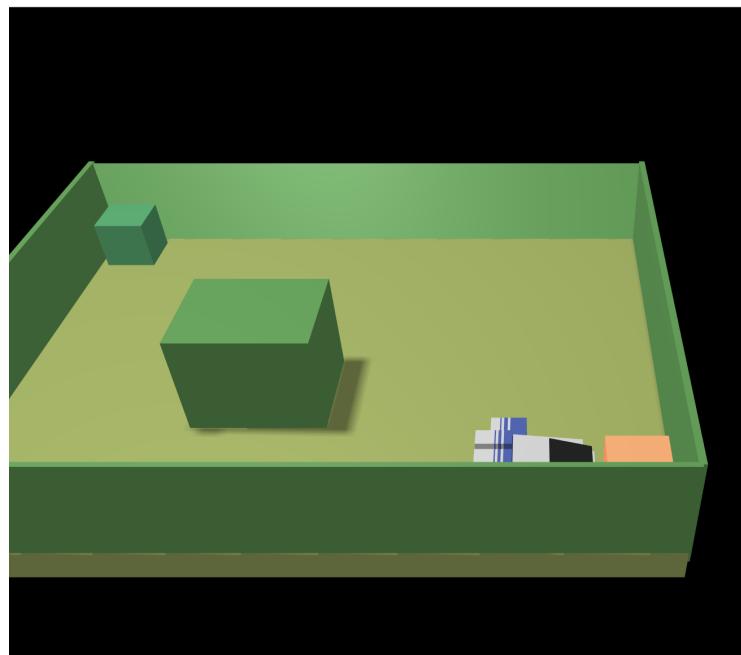


Figura 50: Virtual Robot sul secondo sonar

finale una volta che l'utente avrà ottenuto l'autorizzazione è quella mostrata in figura 51.

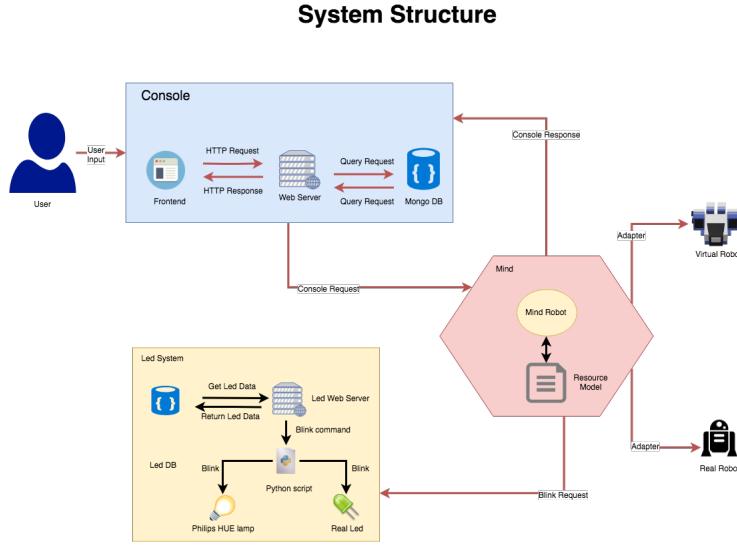


Figura 51: Struttura finale

8.3 Implementazione

8.3.1 Implementazione: Mind .

Durante l'implementazione di questo requisito si è deciso di effettuare un piccolo refactoring al *QActor* della Mind in quanto volevamo dare una separazione più netta tra la logica applicativa e l'esecuzione dei comandi del robot. Perciò sono stati aggiunti due nuovi attori che rimarranno interni al contesto della Mind:

- **delegateexecutor** usato solamente per inviare su MQTT i messaggi per muovere i robot; esso comunicherà a sua volta con la Mind tramite messaggi (ID messaggio: **exec**);
- **autopilot** questo *QActor* è stato creato solamente per motivi implementativi in quanto il metodo statico Java usato per avviare il pilota automatico risulta essere bloccante (se lanciato all'interno della Mind non si sarebbe potuto eseguire nessun altro comando); anch'esso comunica con la Mind tramite messaggi (ID messaggio: **exec**)

Nella Mind è stata necessaria anche un'ulteriore modifica per utilizzare il pilota automatico, ovvero si è dovuto aggiungere nel gestore dei messaggi un controllo

per rilevare il comando - `usercmd(robotgui(auto(X)))`- inviato dalla Console per far avviare l'auto pilot (Fig. 52).

```
onMsg moveRobot : usercmd( robotgui(auto(X)) ) -> {
    [ !? model( type(executor, X), name(Y), value(true) ) ] {
        forward autopilot -m exec : mindcmd( auto(low) );
        demo changeModelItem(leds, NAME, on)
    }
    else
        println("Too hot to work or out of time")
}
```

Figura 52: Rilevamento del comando di auto pilot dalla Console

8.3.2 Implementazione: Auto Pilot (Start command) .

Nell’analisi del problema e nella progettazione abbiamo stabilito che, per stabilire la dimensione della mappa, il robot inizialmente deve effettuare dei movimenti prefissati (vedi Figura 49). Per fare questi movimenti, all’interno della classe Java che descrive l’autopilot, sono stati definiti due metodi:

1. Il primo permette al robot di arrivare fino al primo ostacolo che corrisponde al primo muro della stanza;
2. Il secondo, oltre a permettere al robot di arrivare al secondo sonar, di volta in volta effettua un rotazione a destra e uno step in avanti, in modo tale che possa tracciare meglio la mappa della stanza. In questo modo il **planner** avrà più informazioni sulla stanza e quindi sarà più preciso.

```
public static void forwardToObstacle(QActor qa) throws Exception {
    boolean obstacleDetected = false;
    while (!obstacleDetected && !stopAutoPilot) {
        System.out.println("Value of Autopilot: " + autoPilot.stopAutoPilot);
        moveRobot(qa, "w", true);
        obstacleDetected = isObstacleDetected(qa);
        if (obstacleDetected) {
            clearObstacle(qa);
            String obstacleType = getObstacleType(aiutil.getCurrentDirection());
            aiutil.doMove(obstacleType);
        }
    }
}
```

Figura 53: Primo Metodo

Dopo aver realizzato questi due metodi ci siamo resi conto che i sonar erano nel nostro caso inutili, in quanto le verifiche sul fatto che la mossa generata dal planner fosse applicabile o fosse una mossa che portava contro un ostacolo fisso venivano fatte prima di muovere realmente il robot (Fig. 55). Questo ci consentiva, in caso di collisione, di aggiornare la mappa e quindi di richiedere al planner la prossima mossa valida.

Una volta realizzati questi metodi l’autopilot dovrà semplicemente richiamarli in sequenza (Fig. 56) .

```

public static void traceMap(QActor qa) throws Exception {
    boolean obstacleDetected = false;
    String obstacleType = "";
    moveRobot(qa, "a", true);
    while (!obstacleType.equals("obstacleOnRight") && !stopAutoPilot) {
        System.out.println("Value of Autopilot: " + autoPilot.stopAutoPilot);
        moveRobot(qa, "w", false);
        obstacleDetected = isObstacleDetected(qa);
        if (obstacleDetected) {
            clearObstacle(qa);
            obstacleType = getObstacleType(aiutil.getCurrentDirection());
            aiutil.doMove(obstacleType);
        } else {
            aiutil.doMove("w");
            moveRobot(qa, "d", true);
            moveRobot(qa, "w", false);
            obstacleDetected = isObstacleDetected(qa);
            if (obstacleDetected) {
                clearObstacle(qa);
                obstacleType = getObstacleType(aiutil.getCurrentDirection());
                aiutil.doMove(obstacleType);
                moveRobot(qa, "a", true);
            }
        }
    }
}

```

Figura 54: Secondo Metodo

```

public static void cleanRoom(QActor qa) throws Exception {
    boolean obstacleDetected = false;
    List<Action> actions = aiutil.doPlan();
    while (!actions.isEmpty() && !stopAutoPilot) {
        moveRobot(qa, actions.get(0).toString(), false);
        obstacleDetected = isObstacleDetected(qa);
        if (obstacleDetected) {
            clearObstacle(qa);
            /*Verifica della tipologia di ostacolo*/
            if(it.unibo.utils.avoidObstacle.isStatic(qa)) {
                String obstacleType = getObstacleType(aiutil.getCurrentDirection());
                aiutil.doMove(obstacleType);
            }else
                aiutil.doMove(actions.get(0).toString());
        } else {
            aiutil.doMove(actions.get(0).toString());
        }
        actions = aiutil.doPlan();
    }
}

```

Figura 55: Metodo che utilizza il planner

```

public static void start(QActor qa) {
    try {
        init(qa);
        System.out.println("Start auto pilot");
        aiutil.initAI();
        aiutil.cleanQa();
        System.out.println("===== initial map");
        aiutil.showMap();

        // Static moves
        forwardToObstacle(qa); ← 1
        traceMap(qa); ← 2

        // AI moves
        cleanRoom(qa); ← 3

        System.out.println("===== map after clean");
        aiutil.showMap();

    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

Figura 56: Start AutoPilot

8.3.3 Implementazione: Ostacoli .

L'implementazione del meccanismo per evitare e categorizzare gli ostacoli richiede l'interazione tra 3 componenti del sistema:

- La Mind sarà responsabile di intercettare gli eventi emessi su MQTT dai robot relativi al rilevamento di un ostacolo (`sonarDetect` nel caso del robot virtuale e `realSonarDetect` nel caso del robot reale).

Una volta che la Mind avrà analizzato l'evento ricevuto dovrà modificare il Resource Model mediante il comando `demo changeModelItem(CATEG, NAME, VALUE)`, come è evidenziato dalla figura 57;

- Il Resource model sarà responsabile di aggiornare la base di conoscenza del sistema in maniera da consentire agli altri componenti di verificare la presenza o meno di ostacoli. Per farlo, quando verrà invocato dalla Mind il predicato `changeModelItem(CATEG, NAME, VALUE)`, si andrà a modificare il valore del fatto `model(type(sensor, obstacle), name(sonar), value(no))`. inserendo il valore `yes` nel caso di **Robot Virtuale**, mentre nel caso di **Robot Reale** il valore `yes` verrà inserito solo se la distanza da un ostacolo è minore di una certa soglia (40cm), come si può notare nella figura 58;
- Un metodo statico custom, che viene invocato dalla funzione di pilota automatico, deve verificare se l'ostacolo rilevato è statico o mobile; questo metodo controllerà per tre volte, a distanza di un certo intervallo, se l'ostacolo è ancora presente. Ad ogni iterazione il metodo prova ad avanzare di uno step, richiede al Resource model se ha trovato un ostacolo verificando il fatto `model(type(sensor, obstacle), name(sonar), value(VALUE))`. Se `VALUE` ha valore `yes` allora si reimposta il valore a `no` e si ripete il ciclo altrimenti ferma l'iterazione e comunica che l'ostacolo è dinamico. Se l'iterazione viene ripetuta per tre volte l'ostacolo verrà considerato statico (Fig. 59).

```

Plan handleSonarChange resumeLastPlan [
    printCurrentEvent;
    onEvent sonar : sonar (sonar1, soffritti, DISTANCE) -> demo changeModelItem(sonarVirtual, sonar1, DISTANCE);
    onEvent sonar : sonar (sonar2, soffritti, DISTANCE) -> demo changeModelItem(sonarVirtual, sonar2, DISTANCE);
    onEvent sonarDetect : sonarDetect (TARGET, soffritti) -> demo changeModelItem(sonarRobot, sonarVirtual, TARGET);
    onEvent realSonarDetect : realSonarDetect(sonarReal, DISTANCE) -> demo changeModelItem(sonarRobot, sonarReal, DISTANCE)
]

```

Figura 57: Gestione degli eventi `sonarDetect` e `realSonarDetect`

```

changedModelAction( sonarVirtual, NAME, V):- output(modelChanged(sonarVirtual, name(NAME), value(V))).

changedModelAction( sonarRobot, sonarVirtual, V):- changeModelItem(obstacle, sonar, yes).

| changedModelAction( obstacle, sonar, V).

, changedModelAction( sonarRobot, sonarReal, V):-
    minDistance( MIN),
    eval( let, V, MIN),
    realRobotObstacle(),
    sendMsg( mindrobot, resourceChangeMsg, resourceChangeMsg(sensor, sonarRobot, sonarReal, V ) ),
    changeModelItem(obstacle, sonar, yes).

, changedModelAction( sonarRobot, sonarReal, V):-
    changeModelItem( obstacle, sonar, no).

```

Figura 58: Resource model: modifica del fatto `model(type(sensor, obstacle), name(sonar), value(no)).`

```

public static boolean isStatic(QActor qa) throws Exception {
    boolean end = false;
    int counter = 0;
    System.out.println("Verifica dell'ostacolo");
    String direction = "";
    while (!end && counter < 3) {
        direction = it.unibo.exploremap.program.autoPilot.directionToString(it.unibo.exploremap.program.aiutil.getCurrentDirection());
        it.unibo.exploremap.program.autoPilot.moveRobot(qa, "w", false);
        it.unibo.exploremap.program.autoPilot.sleepMilliseconds(300);

        /*Verifico se l'ostacolo non è più presente*/
        if(!it.unibo.exploremap.program.autoPilot.isObstacleDetected(qa))
            end = true;
        else {
            /*L'ostacolo è ancora presente quindi faccio un altro tentativo (massimo 3)*/
            it.unibo.exploremap.program.autoPilot.clearObstacle(qa);
            counter++;
        }
    }
    System.out.println("Counter obstacle: " + counter);
    if (counter == 3)
        return true;
    else
        return false;
}

```

Figura 59: Metodo per controllare se un ostacolo è statico

8.3.4 Implementazione: Mappa sulla GUI .

La rappresentazione della mappa è stata fatta non solo all'interno del sistema Robot ma anche nella GUI dell'utente per rendere il tutto più appetibile sul mercato; perciò il metodo del pilota automatico ad ogni nuova mossa effettuata invia lo stato attuale dell'intera mappa al Web Server node che provvederà a salvare la mappa nel suo database e inviare quest'ultima alla GUI Reac. In questo modo sarà visualizzata direttamente sul dispositivo dell'utente in modo da mostrare l'avanzamento del robot passo dopo passo (Fig. 60).

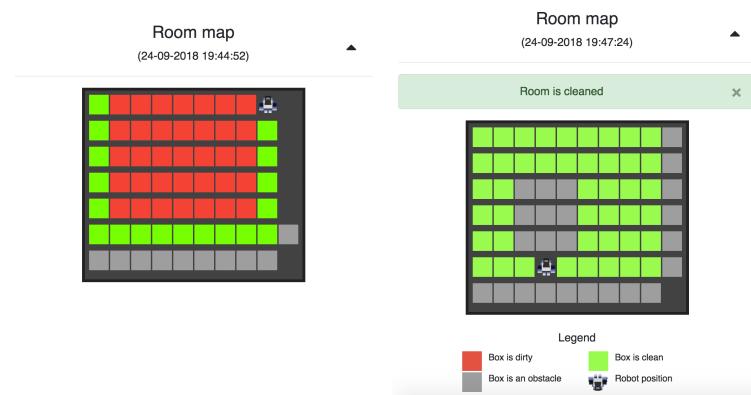


Figura 60: Stanza ancora sporca figura di sinistra, Stanza pulita figura di destra

9 Autori



Figura 61: Angelo Feraudo, Alessandro Staffolani

Link al repository github: <https://github.com/ale8193/robot-cleaner>