Sistema di spedizione automatizzato mediante robot mobili

Laureando
Mattia Pannone

Relatore
Giorgio Grisetti

SAPIENZA
UNIVERSITÀ DI ROMA

SAPIENZA
UNIVERSITÀ DI ROMA

Sistema di spedizione automatizzato mediante robot mobili

Facoltà di Ingegneria dell'informazione, informatica e statistica
Dipartimento di Ingegneria informatica, automatica e gestionale
Corso di laurea in Ingegneria informatica e automatica

Mattia Pannone
Matricola 1803328

Relatore
Giorgio Grisetti

A.A. 2019-2020

Alla mia famiglia che mi sostiene sempre.
A due persone speciali che se ne sono andate
troppo presto, ma so che in questo giorno
per me importante gioiranno con me.

Indice

Introduzione

Questo progetto, come da titolo, si basa su un sistema di spedizione attraverso piattaforme robotiche mobili.
L'intento è assegnare degli obiettivi ad robot e fare in modo che li raggiunga con successo; anche se il titolo fa riferimento a più robot mobili, viene affrontato l'argomento sempre riferendosi ad un singolo robot in quanto i processi eseguiti su uno valgono per tutti. Per comunicare degli obiettivi ad un robot esistono diversi metodi, nel particolare di questo progetto si vuole sviluppare un'applicazione in grado di comunicare con diversi robot attraverso una rete locale con tecnologia wireless. L'intero lavoro riguardante la programmazione e i test dei robot viene svolto in un ambiente di simulazioni per problematiche riguardanti tempi e costi.
Un altro obiettivo è quello di rendere questo progetto il più scalare e modulare possibile, in modo cioè da implementare un'idea che poi è sviluppabile in diversi campi con poche e semplici modifiche, mantenendo di fatto simile la struttura del sistema.

Questa relazione sarà suddivisa in tre capitoli: nel primo si fa una panoramica su tutti gli strumenti utilizzati per la programmazione di un robot, approfondendo alcuni aspetti più importanti; nel secondo si vedrà la struttura del progetto per come è stato pensato, ovvero l'idea ed il funzionamento di tutti i componenti che lo compongono; infine il terzo capitolo affronta la fase di test effettuata nell'ambiente di simulazione per poi vedere alcuni casi in cui possono essere utili robot mobili prima, ipotesi aggiuntive che non sono state prese in considerazione nel progetto iniziale poi.

Capitolo 1:
La tecnologia di base: ROS & Componenti

In questo capitolo vedremo gli elementi fondamentali che contribuiscono al funzionamento del sistema robot, a partire dalla piattaforma su cui gira il tutto, una specie di sistema operativo chiamato ROS, per poi proseguire con tutti i componenti che aggiungono e completano le funzionalità del robot stesso. Per funzionalità si intendono, nel caso specifico di questa tesi, come il robot si localizza nell'ambiente circostante, come si muove in esso e come raggiunge gli obiettivi.
Viene trattato anche il discorso che riguarda l'utilità di progettare e realizzare il tutto dapprima in un ambiente di simulazione, infatti, come chiarito nell'introduzione, l'intero progetto è stato realizzato solo in quest'ultimo ambiente.

1.1 ROS

ROS è l'acronimo di Robot Operating System, è infatti un insieme di framework, ovvero una serie di librerie e strumenti che offrono le stesse funzionalità di un sistema operativo pur non essendolo da un punto di vista tecnico. Come si può intuire dal nome, la parola Robot sta proprio ad indicare che questi strumenti sono utilizzati nelle applicazioni basate sulla robotica.
Il progetto ROS non nasce da un singolo, bensì è un progetto molto grande che ha visto e sta vedendo molti contributi di vari enti di ricerca, infatti la sua licenza è open source e basata sui sistemi Unix-like, non a caso Ubuntu Linux è classificato come il sistema su cui è supportato ufficialmente, mentre in altri sistemi è in via di sperimentazione. Le librerie principali su cui si basa sono roscpp e rospy, con rispettivamente i due linguaggi C++ e Python; ma la sua continua espansione sta portando anche altri linguaggi di programmazione, ad esempio ad oggi può essere integrato anche con Matlab.
Il progetto nasce dall'esigenza di avere un sistema più semplice per la robotica, infatti un sistema robotico è molto complesso nel suo insieme dipendendo dalle molteplici funzionalità di hardware e software che solitamente venivano integrate in un unico modulo.

Con ROS la programmazione diventa più semplice in quanto permette di progettare e realizzare diversi moduli per i diversi servizi forniti dal robot, in questo modo se ad esempio c'è un problema con una funzionalità, è molto più probabile che il sistema non si blocchi all'improvviso nel completo della sua efficienza, mentre nel caso di una programmazione più centralizzata questo rischio è moto alto; la programmazione con ROS risulta dunque più scalabile e flessibile, un esempio pratico è che permette a diversi organi di lavorare contemporaneamente su moduli diversi che poi potranno scambiarsi messaggi nonostante possano lavorare in modo indipendente l'uno dall'altro.
Partendo proprio da quest'ultimo concetto per il quale si può lavorare per moduli diversi, andiamo ora a vedere gli aspetti tecnici che permettono questo tipo di programmazione.
Innanzi tutto ROS, proprio come un sistema operativo, è esso stesso un software che gira su un calcolatore, dunque viene lanciato un nodo detto master, il quale gestisce tutti gli altri nodi che verranno lanciati successivamente e che contengono le varie funzionalità. Nodo è proprio il nome con cui in ROS si indicano i vari processi, indipendenti l'uno dall'altro, che girano ed eseguono i vari servizi e fondamentalmente non sono altro che programmi scritti in linguaggi sopra citati con una certa struttura che vedremo più avanti.
Questi nodi, proprio come i diversi programmi in un sistema operativo, si possono scambiare informazioni. Queste informazioni sono dette messaggi e sono un'altra componente fondamentale, rappresentati da una struttura dati con campi di diversi tipi, come in una struct in C, scritti in un file a parte (quindi non dichiarati nei nodi come variabili) che ha estensione .msg, se ad esempio dobbiamo far scambiare a due processi i dati di una persona, il file da costruire potrebbe essere "Persona.msg" con la seguente struttura:

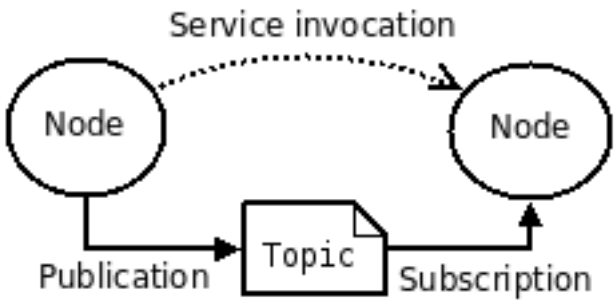
```
_____________-
| string nome       |
| string cognome  |
| uint8 età          |
_____________-
```

I messaggi quando sono inviati e ricevuti sono raggruppati in flussi che contengono informazioni dello stesso tipo, questi flussi sono chiamati topics e sono indicati con delle stringhe, ad esempio un topic che riguarda messaggi del tipo precedentemente illustrato può chiamarsi "info_persone". Nei nodi i componenti adibiti allo scambio di messaggi

vengono chiamati Publisher che li pubblica ed il Subscriber che li sottoscrive. Un nodo può funzionare sia da Publisher che da Subscriber per topics diversi.
Un altro modo che hanno i nodi per scambiarsi informazioni è richiedere ed eseguire servizi, ovvero un nodo server esegue un servizio ed un nodo client ne sfrutta i risultati. A differenza dei topics che vengono pubblicati e successivamente sottoscritti da chi li richiede, un servizio è più diretto in quanto viene chiamata una funzione che restituisce i risultati a chi li chiede, in particolare un client effettua una chiamata al server e si mette in attesa che quest'ultimo risponda con i risultati. Come i messaggi, anche i servizi sono definiti in dei file in cui sono dichiarati i campi che servono per l'elaborazione ed i campi per i risultati, in questo caso l'estensione del file è ".srv". Si può pensare ad un semplice e banale servizio che richiede che effettua una somma, definito nel file "Somma.srv":

```
_____________
| int64 a          |
| int64 b          |
|- - - - - - - - - - |
| int64 somma |
_____________
```

In questo caso il servizio viene passato tra client e server proprio come accade con i messaggi, in questo caso in uno o più nodi vengono dichiarati il ServiceServer che esegue uno specifico servizio e il ServiceClient che lo chiamerà.
Uno schema di comunicazione molto generalizzato può essere rappresentato come segue:



Esempio di comunicazione tra nodi

Ora vediamo da un punto di vista di programmazione come vengono usati gli strumenti sopra elencati, prendendo come riferimento l'anatomia di un nodo ros, ovvero la struttura base dove per comodità sono elencate le istruzioni principali in un unico nodo, attraverso il linguaggio C++:

```
1  #include <ros/ros.h>
2
3  void my_callback(MsgType *msg){
4      //do something
5  }
6
7  bool myservice(ServiceType::Request &req,
8                 ServiceType::Response &res){
9
10     //dosomething
11 }
12
13 int main(int argc, char **argv){
14
15     ros::init(argc,argv,"node_name");
16     ros::NodeHandle n;
17
18     ros::Publisher pub;
19     ros::Subscriber sub;
20     ros::ServiceClicet client;
21     ros::SrviceServer server;
22
23     pub = n.advertise<MsgType>("topic_name",1);
24     MsgType msg;
25     pub.publish(msg);
26
27     sub = n.subscribe<MsgType>("topic_name",1,my_callback);
28
29     server = n.advertiseService("service_name", my_service);
30
31     client = n.serviceClient<ServiceType>("service_name");
32     ServiceType srv;
33     srv.request = //...
34     client.call(srv);
35
36     ros::spin();
37 }
```

La riga 1 ovviamente è la più semplice, ma se vogliamo anche la più importante in quanto è necessaria per usare tutto il resto, infatti importando la libreria di ros avremo a disposizione tutti i suoi strumenti.
Attraverso le righe 15 e 16, rispettivamente andiamo ad inizializzare il nodo ed a creare la variabile attraverso la quale potremo accedere alle funzionalità di quel nodo.

Nelle righe 18, 19, 20 e 21 inizializziamo le componenti di cui si è parlato sopra.
Alla riga 23 viene inizializzato il Publisher con messaggi di tipo indicato nelle parentesi triangolari; il primo parametro riguarda il nome che prenderà il topic, quindi per accedere a quei messaggi successivamente occorrerà sottoscrivere sul topic dello stesso nome. Il secondo parametro riguarda la dimensione della coda di pubblicazione, ovvero quanti messaggi possono attendere in coda per essere pubblicati prima di essere scartati. Nella riga 24 viene dichiarata la variabile messaggio che può essere riempito/modificato accedendo ai suoi campi attraverso la notazione “msg.campo”. Infine nella riga 25 con la funzione publish() viene pubblicato il messaggio.
Alla riga 27 viene inizializzato il Subscriber con il tipo di messaggio da sottoscrivere tra parentesi triangolari, mentre tra parentesi tonde abbiamo il nome del topic sul quale lavorare; come secondo argomento la dimensione della coda dove verranno accodati i messaggi ricevuti se essi possono essere elaborati abbastanza velocemente; infine c’è il nome della funzione da invocare non appena viene ricevuto un messaggio. In questo caso la funzione, definita dalla riga 3, è di tipo void e prende come parametro un puntatore al messaggio sottoscritto, il quale sarà elaborato.
Nella riga 29 abbiamo invece l’inizializzazione di un server e la sua pubblicazione su ros; nel primo parametro abbiamo il nome del servizio attraverso il quale potremo richiamarlo; nel secondo parametro c’è il nome della funzione invocata quando il client effettuerà la chiamata. In questo caso la funzione è definita dalla riga 7, è di tipo booleano e prende come parametri il riferimento ai campi dichiarati nel file .srv, in particolare con “Request” vengono presi i parametri nella prima parte del file che indicano i campi sui quali solitamente sono memorizzati i valori sui quali lavorare, con “Response” vengono presi i parametri che nel file .srv si trovano dopo la linea tratteggiata ed è dove vengono memorizzati i valori di ritorno.
Nella riga 31 c’è l’inizializzazione del client attraverso l’unico parametro che indica il nome del servizio al quale si vuole accedere. Nella riga 32 si crea la variabile associata al servizio voluto e si possono riempirne i campi, come nella riga 33, accedendo alla request e successivamente ai campi disponibili.Successivamente alla riga 34 si effettua la chiamata al servizio passando come unico parametro la variabile appena creata e modificata.
La funzione chiamata alla riga 36 indica semplicemente che quel nodo sarà eseguito continuamente.

Questi elencati sono i metodi principali attraverso un nodo ros è in grado di funzionare, lavorando con messaggi e servizi. Ovviamente esistono sia altre funzioni della libreria ros, nonché le funzione di tutte la librerie C attraverso i quali si possono realizzare le varie funzionalità di un robot, interagendo con i vari sensori ed attuatori disponibili. Nel seguito saranno illustrate le funzioni implementate nel caso del progetto di questa tesi.

1.2 Simulazione e realtà

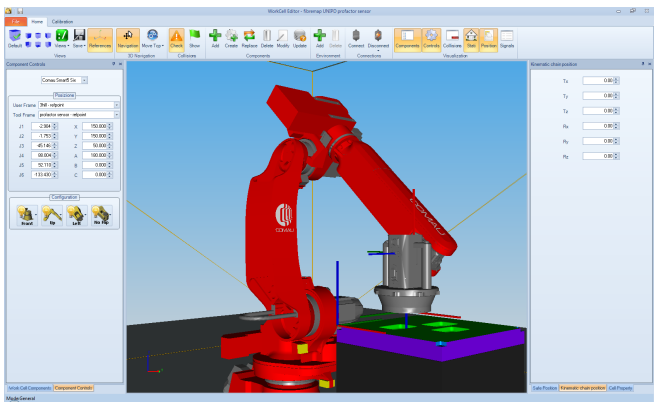
Si precisa che il progetto in oggetto viene è stato svolto completamente in un ambiente di simulazione, sia per problematiche riguardanti il tempo che per problematiche riguardanti i costi. Tuttavia in questa sezione voglio approfondire in modo più ampio le differenze tra lavorare in simulazione e lavorare fisicamente nella realtà su progetti di questo tipo ed il perché in ogni caso simulare la realtà è sempre il primo passo da fare. Approfondiamo poi l'ambiente di simulazione usato per questo progetto, messo a disposizione da ROS.

1.2.1 Perché simulare la realtà

La simulazione è uno strumento importante che permette, a seconda dell'accuratezza, di avvicinarsi alla realtà in modo asintotico, raccogliendo tutti i dati e le caratteristiche più importanti per capire dove migliorare il modello prima di effettuare i primi test nella realtà. Infatti la simulazione non esonera dall'effettuare varie prove prima di avere un prodotto e/o servizio funzionante, tuttavia permette tra le varie cose di risparmiare su due dei fattori più importanti nella progettazione di qualcosa, ovvero i tempi ed i costi; queste due caratteristiche sono unite da una terza caratteristica che non sono altro che gli obiettivi e da queste tre proprietà ne consegue una quarta che rimane al centro per tutta la durata del progetto: la qualità. Questi aspetti sono rappresentati dalla seguente figura, ben nota nell'ambito del project management:

Triangolo del project management

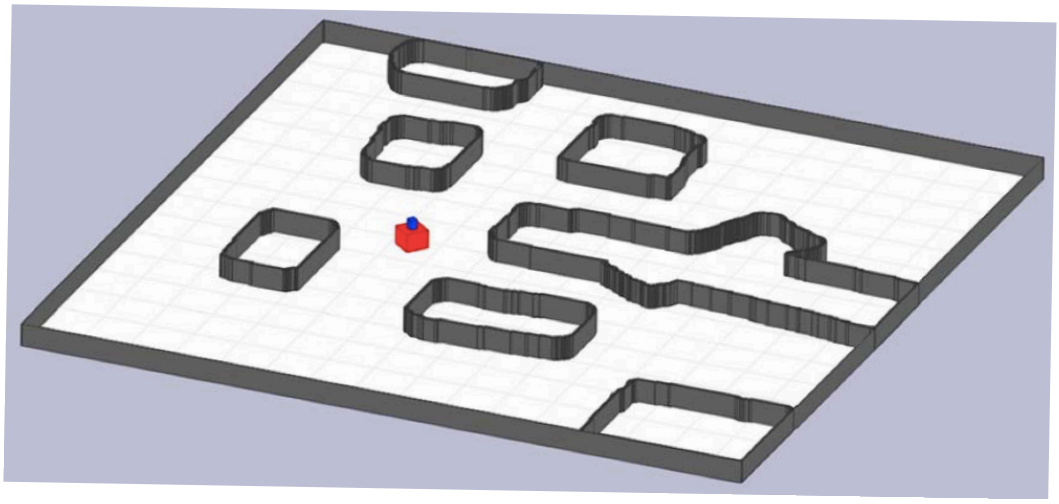
Questa figura rappresenta al meglio le peculiarità elencate in quanto rappresentate attraverso un triangolo equilatero, il ché sta proprio a significare che ognuna è legata all'altra in modo imprescindibile: la qualità è solitamente la caratteristica da mantenere costante sempre, mentre costi, tempi ed obiettivi possono variare, ma la variazione di una porterà alla modifica delle altre due.
La simulazione può essere di vari tipi a seconda dei vari progetti nei diversi ambiti di scienza e ingegneria.
I modelli matematici sono un perfetto esempio oltre a essere alla base della quasi totalità delle simulazioni, un esempio che può venire subito in mente è lo studio dell'andamento di una curva di contagi in una situazione di pandemia, oppure un modello matematico con il quale si può studiare un comportamento fisico; ci sono le simulazioni in ambito architettonico, come i plastici che permettono di presentare ed avere la visione completa di un progetto edile prima della sua costruzione, oppure le strutture per studiare le sollecitazioni in caso di eventi sismici; esistono simulatori di guida areonautica per addestrare i futuri aviatori prima del loro primo volo; l'ambito spaziale è sicuramente un altra categoria che fa uso abbondante di simulazioni, in quanto è facilmente intuibile che non si può rischiare di mandare un veicolo nello spazio senza aver almeno simulato, considerando i tempi ed i costi delle missioni spaziali.
Inoltre molte delle applicazioni oggi sul mercato si basano su applicazioni automatizzate, esempio pratico la guida autonoma; dunque vedere come si comporta un agente automatizzato in un ambiente virtuale prima di poterlo testare, può risultare molto comodo.
Le simulazioni hanno sempre fatto parte della storia dell'uomo, ma sicuramente ad oggi sono facilitate dal fatto che con il supporto dei calcolatori più moderni, esse possono essere effettuate da software appositi per le categorie specifiche, oltre a essere anche più precise grazie alla velocità di calcolo dei diversi dati forniti.



1.2.2 L'ambiente di simulazione: stage_ros

In questo specifico progetto è stato usato un ambiente di simulazione messo a disposizione da ros, stage.
Stage è un ambiente di simulazione di robot, in particolare mette a disposizione un mondo virtuale popolato da robot mobili con diversi sensori e diversi oggetti che possono essere rilevati, come i muri. Oltre a simulare gli oggetti fisici appena menzionati, questo ambiente mette a disposizione i diversi messaggi ed i rispettivi topic che possono essere monitorati e manipolati attraverso la programmazione e quindi creare servizi per i robot. In particolare l'ambiente, dal punto di vista di ros non è altro che un nodo, un processo che pubblica e sottoscrive topic relativi a vari messaggi, come ad esempio l'odometria, la posizione ed il laser. Questo ha permesso di risparmiare tutta la parte di lavoro riguardante il singolo robot, ovvero costruzione e programmazione, nonché la parte di lavoro riguardante l'ambiente in cui far muovere l'agente robotico, quindi topic riguardanti l'ambiente circostante e le mappe (le quali saranno approfondite in seguito), quindi tornando ai vantaggi della simulazione, un altro è che si possono provare servizi riguardanti la navigazione anche se momentaneamente non si ha a disposizione un ambiente reale.
Stage stesso è una libreria software scritta in C++ e ci sono diversi metodi per usarlo uno dei quali, essendo una libreria, è quello di includerla nel proprio codice e "smanettare" modificando a proprio piacere e secondo i propri scopi l'ambiente. Un altro metodo è usarlo come plug-in per Player, un software che fa da interfaccia per uno o più robot, o meglio può essere definito come un livello di astrazione del robot in quanto tutti i dispositivi sono astratti in un insieme di interfacce predefinite. L'ultimo metodo, quello utilizzato in questo progetto, è lanciare stage come un programma a sé, con la sua interfaccia e con le funzioni precedentemente descritte.
Come detto presenta molti vantaggi: è molto realistico per la maggior parte degli obiettivi che in genere si vogliono studiare, comprende già i sensori più comuni, è compatibile con tutti i sistemi Unix-like, diversi robot possono condividere lo stesso mondo. Ma ci sono anche vantaggi non legati direttamente a caratteristiche tecniche nell'ambito robotico: la licenza del software è libera, c'è una comunità attiva di utenti e sviluppatori che contribuiscono a risolvere problemi, è una piattaforma oramai ben conosciuta, il ché contribuisce ad allargarne la compatibilità.
Oltre a caricare i topics del robot, come detto, vengono caricati i topics relativi all'ambiente circostante, il quale viene specificato nella command-line come parametro nel comando per lanciare stage. In particolare, tale

ambiente prende il nome di mondo e non è altro che un file in cui sono specificati tutte le cose che si trovano nell'ambiente, in particolare il numero di robot, i muri, gli ostacoli e così via. Questo file ha un'estensione ".world" e potrebbe essere modificato in base alle proprie esigenze, tuttavia stage stesso ne mette a disposizione più di uno.
Un esempio di come appare l'ambiente in stage è fornito dalla seguente immagine:



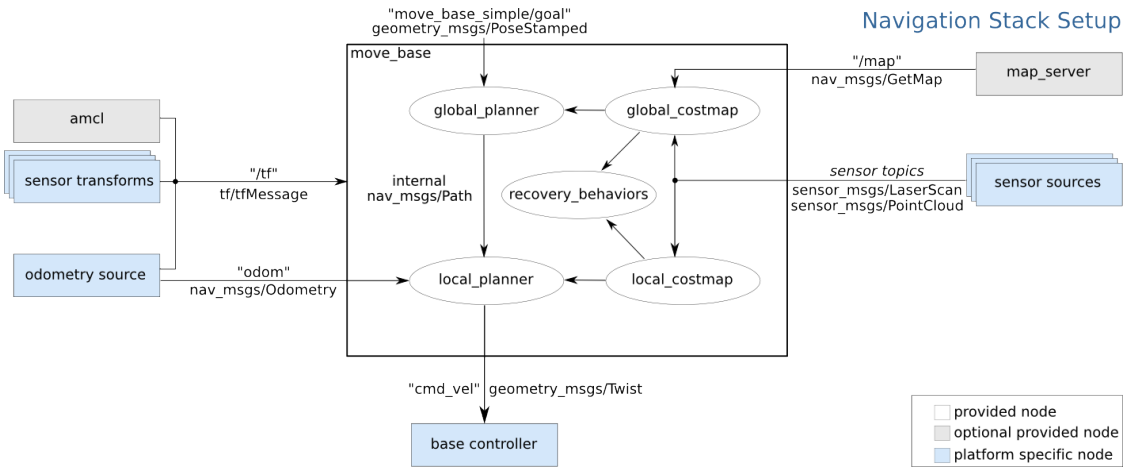
Viene specificato che oltre ad poter essere programmato, il robot può essere gestito attraverso un nodo ros che permette di imporgli comandi di velocità, sia lineare che angolare, nonché comandi di direzione.
L'ambiente viene lanciato con "rosrun stage_ros stage_ros nome_mondo.world".

1.3 Navigazione di un robot

Dopo aver configurato l'ambiente ros e l'ambiente di simulazione, uno dei primi compiti da impartire ad un robot mobile è proprio la navigazione, se non altro non sarebbe un robot mobile, bensì fisso, come il braccio di una catena di montaggio. La navigazione è dunque un fattore importante e non semplicissimo, in quanto dobbiamo insegnare al robot dapprima dove si trova, poi, in base al fatto se ha già memorizzato una mappa in sé oppure no, deve muoversi lentamente per esplorare e dopo aver la mappa chiara dell'ambiente in cui si muove, deve poter navigare

evitando ostacoli che nella mappa iniziale non c'erano (ad esempio persone che gli attraversano la strada). Un componente fondamentale è il laser, grazie al quale si può scansionare l'ambiente circostante.
Andiamo nel seguito a vedere i componenti base che permettono la navigazione ad un robot, approfondendo quelli utilizzati in questo progetto.

1.3.1 Navigation stack

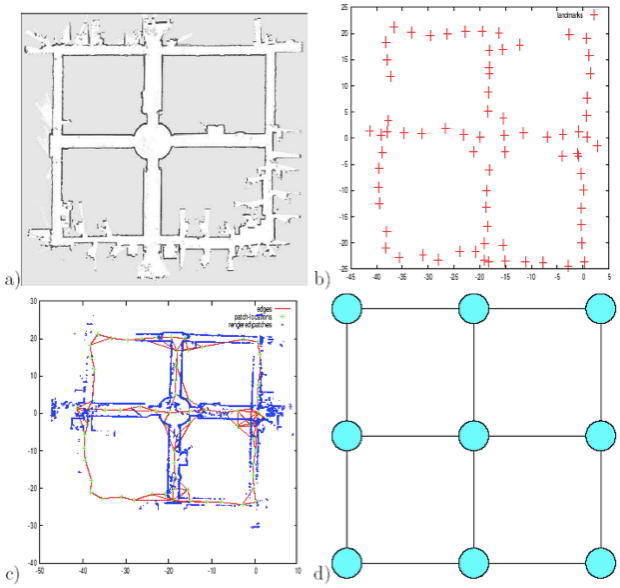


Lo schema rappresentato, mostra i componenti fondamentali della navigazione stack, grazie al quale un robot può risolvere tre problemi in particolare, la mappatura dell'ambiente, la sua localizzazione e la pianificazione si un percorso. Il quadrato centrale "move_base" rappresenta il fulcro dove questi tre problemi vengono elaborati ricevendo varie informazioni, ad esempio, come si può vedere dalle frecce entranti c'è l'obiettivo che viene dato, la mappa dell'ambiente in cui si trova, informazioni relative ai sensori, mentre vengono forniti in uscita i comandi che passano al controllore della base il quale impone ad esempio velocità e direzione.
Il nodo "global planner" si occupa di costruire un percorso basandosi su un algoritmo che implementa, che in genere è l'algoritmo di Dijkstra per il percorso minimo; il "local_planner" si occupa di trasformare il percorso trovato in punti più piccolo e vicini al robot, in modo da poterlo modificare velocemente ad esempio nel caso di ostacoli improvvisi; proprio per gli ostacoli si occupano i nodi "global_costmap" e "local_costmap", in

particolare il primo per ostacoli già noti come muri e colonne, mentre il secondo per ostacoli improvvisi, infatti riceve continuamente informazioni dai sensori e dopo averli elaborati li passa al “local_planner”.
Altri messaggi importanti per l’elaborazione di un percorso sono quelli riguardanti le trasformate, per questo c’è un albero delle trasformate che va inizializzato correttamente, grazie al quale si può sapere ad esempio a quale distanza si trova da terra la base del robot e come sono orientati i sensori rispetto alla stessa base.

1.3.2 Le mappe secondo gli “occhi” del robot

Approfondiamo ora il discorso relativo alle mappe. Il ragionamento per cui un robot esplora un ambiente e successivamente riesce a localizzarsi in esso è simile a quello di un essere umano, non a caso sul robot girano algoritmi elaborati dall’uomo, tuttavia il robot “vede” l’ambiente in modo diverso in base a diversi sensori tra cui il laser ed i dati odometrici.
Una mappa può essere rappresentata in diversi modi: metrico, topologico oppure ibrido tra i due, come mostrano le seguenti immagini:



Questi algoritmi usati per mappare un luogo e localizzarsi sono detti algoritmi di SLAM: Simultaneous Localization and Mapping. Ci sono due passi principali per far sì che attraverso questi algoritmi il robot possa conoscere la mappa dell’ambiente in cui si trova: il primo è far muovere il robot memorizzando a mano a mano le informazioni ed elaborandole

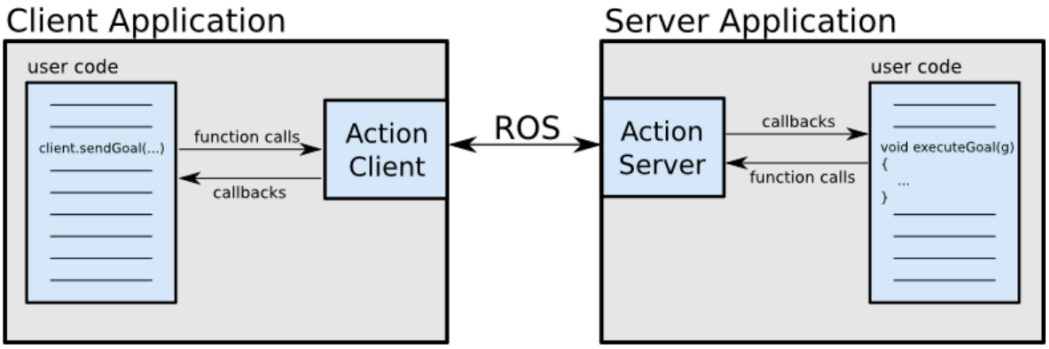
successivamente, il secondo è fornirgli una mappa pubblicandola attraverso un server. Questo server è fornito dal nodo “map_server”, il quale fornisce tutte le informazioni al “global_costmap” (come evidenziato nello schema precedente) pubblicando i relativi topics.
In questo progetto si sono seguiti i seguenti passi per ottenere la mappa dell’ambiente (ricordando che si tratta di un ambiente virtuale fornito dal simulatore stage): attraverso un servizio, fornito sempre da ros, sono stati registrati in un file tutti i topics relativi odometria (“/odom”), al laser (“/scan”) e alle trasformate (“/tf”); dopodiché tale file è stato riprodotto nel mentre era ascoltato dall’algoritmo Gmappig, il quale è l’algoritmo di SLAM messo a disposizione da ros. Infine attraverso il nodo “map_saver” le informazioni raccolte ed elaborate da gmappig hanno salvato due file: un’immagine relativa alla mappa ed un file con estensione “.yaml” che è quello più importante, infatti questo file contiene tutte le informazioni della mappa (es. origine, rapporto tra un pixel ed il mondo reale) ed è il file che viene preso come parametro dal map_server per fornire tali informazioni al robot.
L’immagine della mappa usata per questo progetto e ottenuta attraverso le modalità appena descritte è la seguente:

1.4 Obiettivi del robot

Ottenuta la mappa, uno o più robot mobili possono cominciare a muoversi in essa in diversi modi e possono cominciare a svolgere i compiti per cui vengono programmati in maniera quasi del tutto autonoma. In questo progetto l'obiettivo principale di un robot è quello di spostarsi in punti della mappa ben specifici e tornare alla posizione di partenza. Per questo scopo è stata usata la libreria "Actionlib" che permette di fornire dei goal al robot, restituendo un feedback che fa capire se il compito è stato svolto correttamente oppure no.

1.4.1 La libreria Actionlib

La libreria actionlib fornisce un'interfaccia con le attività che un robot può eseguire, come ad esempio raggiungere un punto, eseguire una scansione, riconoscere un oggetto, ed è basato sul modello client server, un pò come di nodi client e server basilari di ros visti all'inizio. Tuttavia, rispetto a quest'ultimi, i compiti richiesti dal client e svolti dal server vengono chiamati azioni, le quali possono comprendere azioni più durature rispetto ai semplici servizi, inoltre hanno la possibilità di essere monitorati costantemente ricevendo dei feedback e c'è la possibilità di cancellare l'obiettivo fornito.
Lo schema di comunicazione è il seguente:



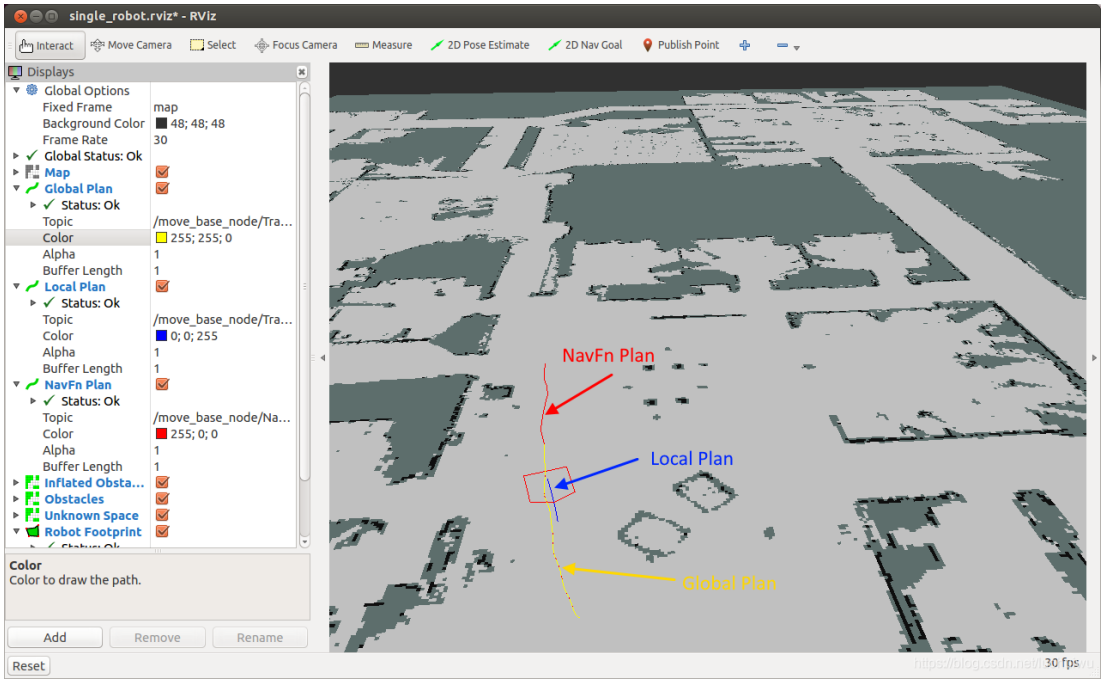
I strumenti messi a disposizione di questa libreria permettono di poter scrivere sia la parte client che quella server, anche se è più diffuso scrivere un client oltre ad essere più facile, mentre si usa scrivere server per compiti più specifici ma molti servizi di base sono già implementati.

Inoltre esistono due tipi di client e due tipi di server che è possibile implementare: i “SimpleActionServer” ed i “SimpleActionClient” che sono quelli più semplici, come dice la parola stessa, da implementare, infatti hanno alcune limitazioni: si può implementare un solo goal, esso deve essere l’unico goal nello stato attivo; mentre ci sono i più complessi “ActionServer” e “ActionClient” che tra le altre cose permettono di avere attive più istanze e dunque svolgere azioni in modo parallelo.
Per quanto riguarda le azioni abbiamo detto che hanno più caratteristiche rispetto ai semplici servizi, tra essi abbiamo: i goal, che sono strutture dati che incapsulano una configurazione per l’obiettivo che deve raggiungere il robot; i feedback che sono aggiornamenti costanti sullo stato corrente dell’azione; i risultati che sono segnali inviati una volta che l’azione è stata completata comunicandone il successo o il fallimento. Ci sono diversi stati in cui un’azione si può trovare, i principali sono: “PENDING”, cioè il goal deve ancora essere processato dal server; “ACTIVE”, il server sta processando il goal; “SUCCEDED”, il goal è stato completato con successo; “ABORTED”, il goal non è stato completato per qualche motivo; “REJECTED”, il goal è stato rigettato dal server senza una richiesta di cancellazione da parte del client; “PREEMPTING”, il processo relativo al goal è stato cancellato dal subentro di un altro goal oppure è stata inviata una richiesta di cancellazione dal client.
In questo progetto è stato implementato un SimpelActionClient con lo scopo di inviare al server una posizione identificata dai punti (x,y) e far raggiungere quel punto dal robot.

1.5 Visualizziamo la simulazione: RVIZ

Come detto all’inizio, tutti gli strumenti usati precedentemente sono spesso usati in un’ambiente di simulazione. Ros tuttavia mette a disposizione uno strumento grafico utilissimo che permette di visualizzare ciò che viene fatto attraverso una rappresentazione tridimensionale. Questo strumento inoltre può essere utilizzato anche per visualizzare i dati dei test reali oltre che delle simulazioni, infatti è un ottimo strumento di debug, grazie anche al fatto di poter scegliere se visualizzare tutto o solo alcuni elementi, infatti la sua interfaccia permette di scegliere tra i topic attualmente pubblicati su ros, quali visualizzare. Inoltre è possibile inviare dei comandi direttamente ai nodi in esecuzione in modo da visualizzarne i risultati senza modificare i diversi programmi.

L'interfaccia è semplice ed intuitiva, come si può vedere dalla seguente immagine:



Si può osservare nella barra laterale sinistra la lista dei topic scelti da visualizzare oppure aggiungerne altri attraverso il comando "Add"; nel riquadro destro abbiamo la visualizzazione vera e propria, ad esempio in questo caso si possono vedere la mappa ed il percorso locale e globale ottenuto; in alto c'è la barra con varie voci del menu, ad esempio con "2DNav Goal" si può inserire un punto nella mappa dove far arrivare il robot.

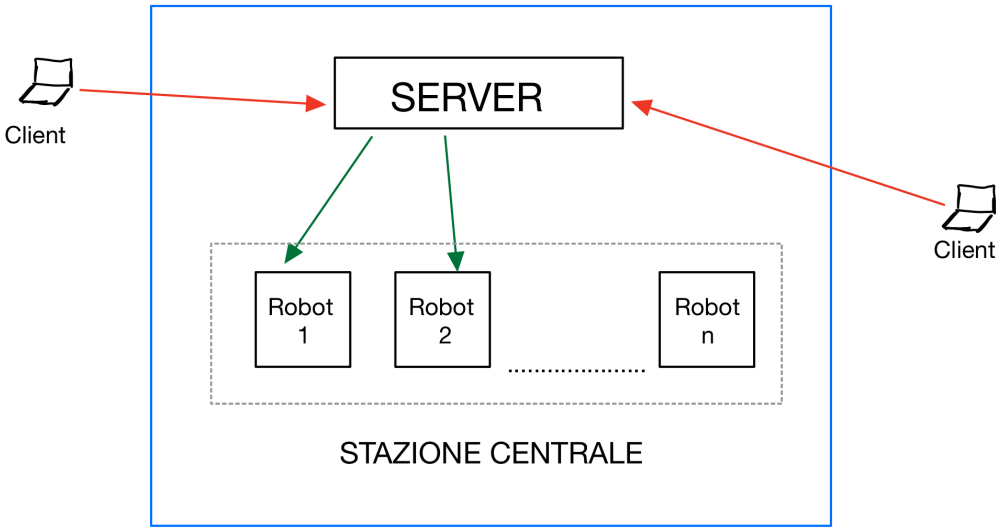
Capitolo 2:
Struttura del sistema automatizzato

Nel primo capitolo abbiamo visto tutti gli strumenti base e necessari utilizzati per la programmazione di un singolo robot. In questo capitolo andremo nel dettaglio, nel cuore del progetto, spiegando in cosa consiste un sistema automatizzato per la consegna utilizzando appunto uno o più robot mobili. Affrontiamo prima la programmazione del singolo robot sviluppata per farlo adempiere ai doveri in oggetto; poi viene spiegata l'applicazione sviluppata per permettere di gestire il sistema e interfacciarsi con il robot; infine è affrontata la struttura generale a livello di rete, ovvero a livello di comunicazione.
Si ricorda che il progetto è stato svolto in un ambiente di simulazione.

2.1 L'idea

Come da titolo, l'idea si basa su un sistema composto da uno o più robot in grado di muoversi e raggiungere degli obiettivi ben precisi. Oltre a muoversi si prevede che il robot sia in grado di trasportare oggetti. In questa relazione non è definito cosa debba trasportare per due principali motivi: il primo è che essendo sviluppato in simulazione, fisicamente non viene trasportato niente; il secondo è che può essere adattato a diversi scopi e dunque adattare di conseguenza la struttura del robot, mentre il software rimane sempre lo stesso, se non per qualche caratteristica da settare in file di configurazione appropriati, ad esempio prima si è parlato dell'albero delle trasformate che riguarda tra le altre cose anche la posizione dei sensori rispetto alla base del robot, dunque se la struttura sarà diversa ed i sensori montati in un modo diverso, bisognerà modificare l'albero delle trasformate.
Un'altra componente importante per la gestione di questo sistema automatizzato, consiste in un'applicazione per comunicare con i robot in modo da imporgli gli obiettivi da raggiungere. Quest'applicazione è stata sviluppata da zero e si è basata su uno degli esempi di uso di questo sistema che sarà spiegato nel prossimo capitolo, tuttavia come vedremo l'organizzazione del codice di tale applicazione permette di essere modificato facilmente ed essere adattato a diverse esigenze.
L'idea di comunicazione è che l'applicazione si connetta ad un server nel quale ci sono le informazioni necessarie, dopodiché viene contattato il

robot interessato al quale viene comunicato l'obiettivo da raggiungere. L'idea alla base dello sviluppo prevede anche due metodi principali attraverso il quale il robot postino può operare: il primo, chiamato "Spedizione da centrale", consiste nel fatto che il robot parta con il carico da consegnare direttamente dalla stazione centrale in cui sono parcheggiati i robot; il secondo, battezzato "Spedizione su richiesta", consiste che il robot parta dalla centrale e si rechi in un punto dove preleva il carico, poi come seconda operazione si rechi nel punto di consegna ed infine torni da dove è partito.
La struttura in modo generalizzato può essere rappresentata così:

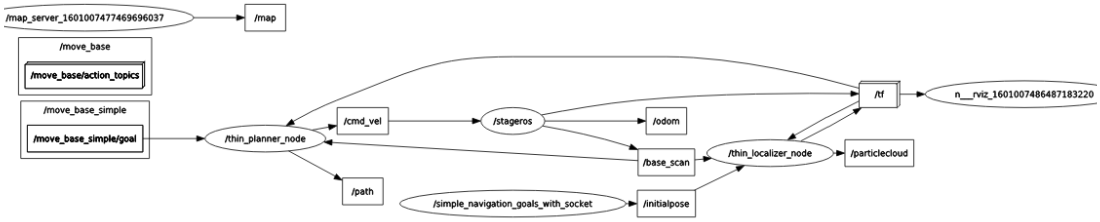


2.2 I robot

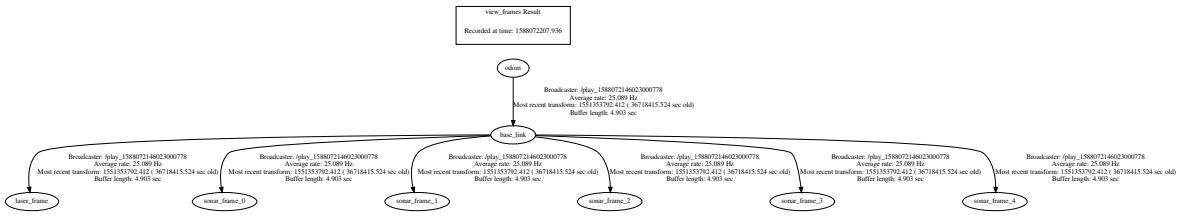
Questa sezione si occupa dei robot, anzi di un robot, considerando il fatto che anche se nel sistema ce ne sono n, ognuno di essi esegue gli stessi processi, quindi la struttura software è uguale per tutti. Abbiamo detto che questa relazione si basa su una simulazione attraverso l'ambiente stage di ros, dunque non vedremo nello specifico il processo ros che elabora ad esempio i dati dei sensori, vediamo invece le parti di codice che danno al robot gli obiettivi e la parte che permette di comunicare con l'esterno, nonché l'elaborazione dei dati che effettua.

2.2.1 I processi in esecuzione

Tornando un attimo al nucleo di ros, nel seguito, attraverso uno schema generato grazie a rqt_graph, uno strumento che ros mette a disposizione per visualizzare i nodi, i topic ed i servizi attualmente in esecuzione, vediamo appunto cosa si c'è in esecuzione sul calcolatore di un robot:



Si possono vedere gli ovali che corrispondono a nodi ros, i quadrati che corrispondono a topic, pubblicati dai nodi che li puntano con una freccia e sottoscritti dai nodi le cui frecce sono entranti. In questa immagine le scritte sono molto piccole, ma si può scorgere in alto a sinistra il nodo "map_server" che pubblica il topic "/map", il nodo centrale "/stageros" che racchiude i principali topic tipici del robot, ovvero pubblica i dati odometrici con "/odom" e quelli relativi al laser con "base_scan", mentre sottoscrive quelli relativi ai comandi per gli spostamenti "/cmd_vel". Come accennato precedentemente ci sono poi i nodi thin_planner e thin_localizer che si occupano della navigazione e ricevono ed emettono diversi topic, in particolare il primo pubblica "/path" che sono informazioni sul percorso calcolato, mentre il secondo pubblica "/particlecloud" che sono informazioni relative ai punti della mappa. Il nodo "/simple_navigation_goal_with_socket" in basso rappresenta il nodo sviluppato per questo progetto. A destra c'è il nodo relativo ad rviz dove viene visualizzato il tutto e che riceve informazioni da "/tf"; proprio su quest'ultimo, che ha informazioni sull'albero delle trasformate, possiamo notare che entrano ed escono molte frecce, infatti come accennato prima sono di rilevante importanza le posizione relative ed assolute di robot e sensori. Possiamo vedere la struttura di un albero delle trasformate nella seguente figura, ottenuta grazie a strumenti di ros che permettono di generarlo e pubblicarlo nel caso ce ne fosse bisogno.



2.2.2 Il codice

Andiamo ora a vedere nello specifico il nodo ros, scritto in C++, realizzato per questo progetto.

```
2  #include <ros/ros.h>
3  #include <move_base_msgs/MoveBaseAction.h>
4  #include <actionlib/client/simple_action_client.h>
5  #include <geometry_msgs/PoseWithCovarianceStamped.h>
6  #include <time.h>
7  #include <iostream>
8
9  #include <arpa/inet.h>
10 #include <netinet/in.h>
11 #include <sys/socket.h>
12 #include <sys/types.h>
13 #include <stdlib.h>
14 #include <stdio.h>
15
16 #include "provaTesi.h"
17
18 #define SERVER_PORT 3000
19
20 typedef actionlib::SimpleActionClient<move_base_msgs::MoveBaseAction> MoveBaseClient;
21
22 int main(int argc, char** argv){
23
24     ros::init(argc, argv, "simple_navigation_goals_with_socket");
25
26     //set initial pose
27     set_initial_pose();
28
29
30     //init socket
31     int sock, lenght, n;
32     socklen_t len;
33     struct sockaddr_in server, client;
34     char buf[4];
35
36     sock = socket(AF_INET, SOCK_DGRAM, 0);
37     if(sock<0){
38         perror("Opening socket");
39         exit(0);
40     }
41
42     lenght = sizeof(server);
43     bzero(&server,lenght);
44     server.sin_addr.s_addr = INADDR_ANY;
45     server.sin_family = AF_INET;
46     server.sin_port = htons(SERVER_PORT);
47
48     if(bind(sock,(struct sockaddr*)&server,lenght)<0){
49         perror("binding");
50         exit(0);
51     }
52     len=sizeof(struct sockaddr_in);
```

Le prime 16 righe riguardano le librerie necessarie per usare tutte le funzioni necessarie, alla riga 20 è dichiarato un Simple Action Client che sarà inizializzato ed usato nel seguito; nel corpo principale viene inizializzato il nodo per ros (riga 24), viene settata la posizione iniziale

rispetto alla mappa (riga 27), tra le righe 31 e 34 vengono inizializzati le variabili necessarie per una socket, la quale viene creata alla riga 36 con successivo controllo se l'operazione è andata a buon fine, fino alla riga 52 vengono effettuate le operazioni necessarie per inizializzare e far funzionare la socket.

La funzione set_inital_pose alla riga 27 è così implementata:

```
6   geometry_msgs::PoseWithCovarianceStamped initPose;
7   ros::Publisher initP;
8
9   void initial_pose(const nav_msgs::Odometry &msg){
10
11      std::cout<<"frame_id :"<<msg.header.frame_id<<"\n";
12      std::cout<<msg.pose.pose.position<<"\n";
13
14      initPose.header=msg.header;
15      initPose.pose.pose.position=msg.pose.pose.position;
16      initPose.pose.pose.orientation=msg.pose.pose.orientation;
17
18      initP.publish(initPose);
19  }
20
21  void set_initial_pose(){
22
23      ros::NodeHandle n;
24      initP = n.advertise<geometry_msgs::PoseWithCovarianceStamped>("/initialpose",1);
25      ros::Subscriber sub = n.subscribe("/base_pose_ground_truth",1,initial_pose);
26
27  }
```

Dichiarata ed implementata tra le righe 23 e 27, essa inizializza un Publisher dichiarato globalmente alla riga 7, per poi sottoscrivere il topic relativo alla vera posizione della base in quel momento data da "/base_pose_ground_truth", chiamando la funzione initial_pose implementata tra le righe 9 e 19. Quest'ultima funzione prende i dati relativi alla posizione mettendoli nella corrispondente struttura e publica tali informazioni con il topic "/initialpose", che fornisce la posizione iniziale al nodo thin_localizer, come si può notare nel grafo precedente.

Successivamente la socket si aspetta di ricevere dei messaggi ed una volta ricevuti li elabora:

```
53    //ricezione ed elaborazione messaggio
54    n=recvfrom(sock,buf,4,0,(struct sockaddr*)&client,&len);
55    if(n<0) {perror("Errore ricezione"); exit(0);}
56    int x = atoi(buf);
57
58    bzero(buf,4);
59    n=recvfrom(sock,buf,4,0,(struct sockaddr*)&client,&len);
60    if(n<0) {perror("Errore ricezione"); exit(0);}
61    int y = atoi(buf);
62
63    //flag
64    bzero(buf,4);
65    n=recvfrom(sock,buf,4,0,(struct sockaddr*)&client,&len);
66    if(n<0) {perror("Errore ricezione"); exit(0);}
67    int flag = atoi(buf);
68
69    std::cout<<"Ricevuto: x = "<<x<<" e y = "<<y<<"\n";
70
```

Le righe 54, 59 e 65 attendono finché non viene ricevuto qualcosa, dopo aver ricevuto, se tale ricezione è andata a buon fine, il messaggio viene elaborato, in particolare, essendo sotto forma di caratteri, essi vengono trasformati in interi, in quanto c'è la necessità di lavorare con numeri interi. In particolare i messaggi ricevuti riguardano i primi due le coordinate x e y del punto in cui si vuole mandare il robot, mentre il terzo riguarda l'opzione riguardante le due modalità di spedizione spiegate prima, a breve è spiegato come esso è utilizzato.

```
70
71    //tell the action client that we want to spin a thread by default
72    MoveBaseClient ac("move_base", true);
73
74    //wait for the action server to come up
75    while(!ac.waitForServer(ros::Duration(5.0))){
76        ROS_INFO("Waiting for the move_base action server to come up");
77    }
78
79    move_base_msgs::MoveBaseGoal goal;
80
81    //first goal
82    goal.target_pose.header.frame_id = "base_link" ;
83    goal.target_pose.header.stamp = ros::Time::now();
84    goal.target_pose.pose.position.x = x;
85    goal.target_pose.pose.position.y = y;
86    goal.target_pose.pose.position.z = 0.0;
87    goal.target_pose.pose.orientation.w = 1.0;
88    ROS_INFO ("Sending goal" );
89    ac.sendGoal(goal);
90
91    //wait result
92    ac.waitForResult();
```

Dalla riga 72 viene inizializzato ed usato il Simple Action Cilent, in particolare proprio alla riga 72 esso viene creato prendendo due argomenti: il primo è il nome del server a cui si connette, ricordando che in questo caso è già implementato, il secondo argomento, un booleano, permette di creare un thread automaticamente se settato a true. Successivamente, riga 79, viene dichiarato un goal di tipo move_base_msgs e successivamente riempito con i dati ricevuti dalla socket x e y, oltre ad un'altra serie di informazioni. Con la funzione sendGoal() alla riga 89, il goal viene inviato al server, mentre alla riga 92 si attende risposta da esso.

```
94    if(ac.getState() == actionlib::SimpleClientGoalState::SUCCEEDED ||
95       ac.getState() == actionlib::SimpleClientGoalState::ABORTED)
96        ROS_INFO("Goal completed" );
97    else{
98        ROS_INFO("The base failed to move for some reason");
99        //cancel goal
100       ac.cancelAllGoals();
101   }
102
103   //set initial pose
104   set_initial_pose();
105
```

Tra le righe 94 e 101 viene controllato lo stato dell'azione eseguita dal server, in caso di successo si continua, altrimenti viene stampato un messaggio di errore ed in ogni caso viene cancellato il goal. Si può notare che alla riga 95 il caso di stato "ABOTRED" è dato come buono, in realtà anche esso fa parte di uno stato di fallimento, ma è stato inserito nella casistica buona a scopo di debug.
Nella riga 104 viene recuperata nuovamente la posizione attuale.

```
105
106   //check flag
107   if(flag==0){
108       x=-11;
109       y=23;
110   }
111   else{
112       n=recvfrom(sock,buf,4,0,(struct sockaddr*)&client,&len);
113       if(n<0) {perror("Errore ricezione"); exit(0);}
114       int x = atoi(buf);
115
116       bzero(buf,4);
117       n=recvfrom(sock,buf,4,0,(struct sockaddr*)&client,&len);
118       if(n<0) {perror("Errore ricezione"); exit(0);}
119       int y = atoi(buf);
120   }
121
```

In queste ultime righe entra il gioco la variabile flag, il terzo messaggio ricevuto dalla socket. Come detto prima fa riferimento alle modalità di spedizione elencate, in particolare: nel caso valesse 0, si è scelto di spedire direttamente da centrale, dunque dopo la consegna effettuata il robot deve tornare alla centrale, dunque le variabili x e y vengono settate direttamente alle coordinate della posizione iniziale; nel secondo caso, in cui si riceve 1, il secondo obiettivo del robot riguarda le informazioni della destinazione (ricordiamo che in questo caso le prime coordinate indicavano il punto di ritiro), dunque il robot si mette in attesa per riceverle.
Dopo queste operazioni viene di nuovo riempita la struttura goal e viene inviata al server, viene atteso il risultato ed infine il ciclo ricomincia da capo. Il codice è simile a quello già visto.

Facendo un breve riassunto, il comportamento del robot da quando viene messo in funzione è il seguente: si mette in attesa dei messaggi dalla socket che gli servono per completare l'obiettivo; una volta ricevuti vengono elaborati e copiati nella struttura goal attraverso la quale viene dato il primo obiettivo; una volta ricevuto il risultato viene riempito il goal per il secondo obiettivo a seconda della modalità di spedizione scelta; infine, completato il secondo obiettivo e tornato in ogni caso alla base centrale, si ricomincia da capo.
Nonostante il tutto è realizzato in un ambiente di simulazione, si può dare uno sguardo alla realtà sottolineando che questi nodi ros, ovvero tutta la parte del robot come la navigazione, può essere eseguita su una scheda Raspberry Pi, in quanto può ospitare un sistema Unix-like ed è inoltre una scheda con poco ingombro, poco consumo e poco costo.

2.3 Interagire con i robot

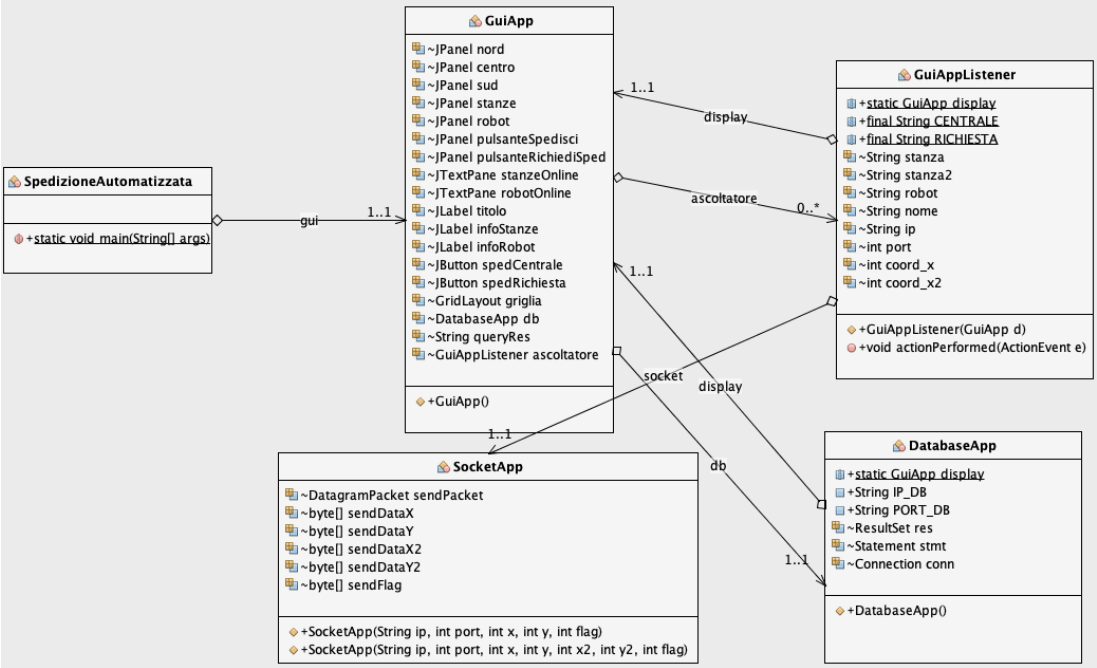
Come detto, questo progetto prevede la possibilità di comunicare gli obiettivi al robot attraverso un'applicazione, abbiamo visto infatti che tra i processi che girano sul robot c'è una socket che viene aperta e si mette in ascolto, quest'applicazione anche implementa delle socket che permettono di collegarsi a diversi robot ed inoltre implementa la connessione ad un database dal quale prende le informazioni necessarie. Vediamo ora nello specifico il duo funzionamento.
Innanzitutto è stato utilizzato il linguaggio java, i quanto il suo eseguibile può girare su qualunque sistema dotato di JVM (Java Virtual Machine) ed è molto flessibile per quanto riguarda la progettazione della sua interfaccia grafica grazie alle API messe a disposizione dalla libreria Swing, nonché per la connessione ad un database attraverso il JDBC, (Java DataBase Connectivity).

2.3.1 L'interfaccia

L'interfaccia progettata è la seguente:

In questo caso si fa riferimento a quattro robot che possono andare in sei diverse stanze; in particolare nel pannello centrale vengono stampati i robot e le stanze disponibili in quel momento (dopo un'interrogazione al database), con un relativo codice, questi codici vanno inseriti nel passo successivo quando si clicca su uno dei due pulsanti in basso "Spedizione da centrale" oppure "Richiedi spedizione". In entrambi i casi vanno inseriti il codice del robot da utilizzare e della stanza in cui mandarlo, solo nel secondo caso serve anche il codice di una seconda stanza. E' previsto che nel momento in cui un robot viene impegnato, esso lo segnali in modo che non risulti momentaneamente disponibile. E' previsto anche che si tenga traccia delle spedizione effettuate, infatti al momento dell'inserimento dei dati viene chiesto nome e descrizione, queste info saranno memorizzate in una tabella del database e consultabili in seguito.
L'applicazione può essere disponibile su qualunque dispositivo connesso alla rete locale, in modo che ognuno può spedire dalla propria postazione senza centralizzare tutto su un unico dispositivo.

2.3.2 Il codice



La figura rappresentata mostra l'organizzazione delle classi, 5 nello specifico, a livello di progettazione attraverso uno schema in UML.
Come detto è stato usato java come linguaggio di programmazione ed essendo un linguaggio orientato agli oggetti, l'applicazione è stata appunto organizzata in classi, dove ognuna svolge un compito specifico, questo rende il codice scalabile e facilmente modificabile per adattarlo a diverse esigenze.
Facendo riferimento allo schema UML si può osservare: la classe "SpedizioneAutomatizzata" che non è altro che la classe principale nel quale è implementato il metodo principale che sarà eseguito al lancio dell'applicazione, essa richiama un oggetto di GuiApp che permette di lanciare l'interfaccia grafica con tutte le funzioni conseguenti. Proprio nella classe "GuiApp" c'è l'organizzazione e l'incapsulamento dei diversi oggetti della libreria Swing di Java per creare l'interfaccia grafica, inoltre vi è il collegamento con altre due classi: "DatabaseApp" e "GuiAppListener". La prima effettua la connessione al database grazie alla quale in GuiApp abbiamo un oggetto di questo tipo grazie al quale effettuare interrogazioni e stampare le informazioni richieste. Nella

seconda sono implementate le azioni da eseguire al click dei pulsanti nell'interfaccia. A quest'ultima classe è collegata "SocketApp", la classe che inizializza la socket per la connessione e la spedizione delle informazioni ai robot.
Vediamo alcuni frammenti di codice che implementano passi importanti. Mentre la classe principale, nel metodo main richiama solo un oggetto di GuiApp e quest'ultima implementa solo il costruttore con banali funzioni per l'incapsulamento dei componenti grafici, vediamo nello specifico alcuni righe che implementano le altre tre classi.

```
18  * @author mattiapannone
19  */
20 public class SocketApp {
21
22     DatagramPacket sendPacket;
23     byte[] sendDataX = new byte[256];
24     byte[] sendDataY = new byte[256];
25     byte[] sendDataX2 = new byte[256];
26     byte[] sendDataY2 = new byte[256];
27     byte[] sendFlag = new byte[256];
28
29     public SocketApp(String ip, int port, int x, int y, int flag) throws SocketException, IOException{
30
31         DatagramSocket clientSocket = new DatagramSocket();
32         InetAddress IPAddress = InetAddress.getByName(ip);
33
34         //invio coordinata x
35         sendDataX = String.valueOf(x).getBytes();
36         sendPacket = new DatagramPacket(sendDataX, sendDataX.length, IPAddress, port);
37         clientSocket.send(sendPacket);
38
39         //invio coordinata y
40         sendDataY = String.valueOf(y).getBytes();
41         sendPacket = new DatagramPacket(sendDataY, sendDataY.length, IPAddress, port);
42         clientSocket.send(sendPacket);
43
44         //invio flag
45         flag = 0;
46         sendFlag = String.valueOf(flag).getBytes();
47         sendPacket = new DatagramPacket(sendDataY, sendDataY.length, IPAddress, port);
48         clientSocket.send(sendPacket);
49     }
```

Classe SocketApp

Nella classe SocketApp, nelle righe 31 e 32 viene inizializzata la socket con l'indirizzo ip passato come parametro, mentre nelle righe successive si effettua un'operazione di spedizione per tre volte, inviando le coordinate richieste ed un flag che indica quale pulsante e quindi quale opzione di invio si è scelta. Viene implementato anche un secondo costruttore, sfruttando l'overloading, che contiene più parametri nel caso vengano spediti più dati. Ovviamente le funzioni di spedizione sono implementate nello stesso modo.

```
11  /**
12   *
13   * @author mattiapannone
14   */
15  public class DatabaseApp {
16
17      public static GuiApp display;
18      public final String IP_DB = "192.168.1.171";
19      public final String PORT_DB = "3306";
20      ResultSet res;
21      Statement stmt;
22      Connection conn;
23
24      public DatabaseApp() throws ClassNotFoundException, SQLException{
25          Class.forName("com.mysql.cj.jdbc.Driver");
26          String url = "jdbc:mysql://"+IP_DB+":"+PORT_DB+"/appspedizioni";
27          String usr = "app";
28          String pwd = "app";
29
30          try{
31              conn = DriverManager.getConnection(url,usr,pwd);
32              System.out.println("Connessione riuscita");
33          } catch (SQLException e) {
34              JOptionPane.showMessageDialog(display,"Si è verificata la seguente eccezione:\n"
35                                      +e+"\nInformare l'amministratore di rete");
36          }
37
38          stmt = conn.createStatement();
39      }
40  }
```

Classe DatabaseApp

La classe DatabaseApp contiene l'indirizzo ip e la porta del server in cui è in esecuzione il database, nonché lo user e la password per accedervi. La connessione avviene alla riga 31 e, in caso di errore, viene gestita un'eccezione; alla riga 38 viene creato un oggetto che permette di accedere alla base di dati consentendo di effettuare delle query.
Importante è ciò che avviene alla riga 25. Lì viene caricato il driver necessario per far funzionare il JDBC; tale driver non è lo stesso per tutti i tipi di database, ma è un file .jar, dunque un eseguibile java, che deve essere scaricato dal sito ufficiale del DBMS in uso e posizionato all'interno del progetto che si sta realizzando, assenza di questo viene generata un'eccezione di tipo ClassNotFoundException.

```
37      @Override
38      public void actionPerformed(ActionEvent e){
39
40          if(e.getActionCommand().equals(CENTRALE)){
41              try {
42                  stanza = JOptionPane.showInputDialog("Inserisci codice stanza dove vuoi consegnare la spedizione");
43                  robot  = JOptionPane.showInputDialog("Inserisci codice robot da usare");
44                  nome   = JOptionPane.showInputDialog("Inserisci il tuo nome");
45                  descrizione = JOptionPane.showInputDialog("Inserisci descrizione");
46
47                  display.db.res = display.db.stmt.executeQuery("SELECT coord_x,coord_y FROM stanza where codice="+this.stanza);
48                  display.db.res.next();
49                  coord_x = display.db.res.getInt("coord_x");
50                  coord_y = display.db.res.getInt("coord_y");
51                  //JOptionPane.showMessageDialog(display, "Hai inserito ("+coord_x+","+coord_y+")");
52
53                  display.db.res = display.db.stmt.executeQuery("SELECT ip_address,porta FROM robot where codice="+this.robot);
54                  display.db.res.next();
55                  ip = display.db.res.getString("ip_address");
56                  port = display.db.res.getInt("porta");
57                  //JOptionPane.showMessageDialog(display, "Hai inserito ("+ip+","+port+")");
58
59              } catch (SQLException ex) {
60                  Logger.getLogger(GuiAppListener.class.getName()).log(Level.SEVERE, null, ex);
61              }
62
63              try {
64                  SocketApp socket = new SocketApp(ip,port,coord_x,coord_y,0);
65                  JOptionPane.showMessageDialog(display, "Inviato con successo");
66              } catch (IOException ex) {
67                  Logger.getLogger(GuiAppListener.class.getName()).log(Level.SEVERE, null, ex);
68              }
69          }
```

Classe GuiAppListener

La classe GuiAppListener è necessaria per eseguire azioni al verificarsi di eventi, essa implementa l'interfaccia "ActionListener" di Java, la quale permette di mettersi in ascolto ed eseguire azioni al verificarsi di eventi, infatti il metodo "actionPerformed" fa parte di tale interfaccia ed è opportunamente modificato attraverso overriding. In particolare in questo frammento di codice è rappresentato cosa succede quando si preme il pulsante "Spedizione da centrale"; tra le righe 42 e 45 vengono richieste le informazioni necessari, successivamente tra le righe 47 e 56 vengono prelevate dal database le informazioni opportune, ovvero: le coordinate della stanza in cui si vuole andare, l'indirizzo ip e la porta su cui si trova connesso il robot scelto ed al quale comunicare le coordinate ottenute.
In questo codice vengono gestiti due tipi di eccezione: la prima è "SQLException" e riguarda la gestione di problemi derivanti dall'interrogazione del database, mentre la seconda è "IOException" che gestisce eventuali errori legati alla socket.
In questa funzione vengono trattati tutti gli eventi che l'applicazione può generare tramite il costrutto "if…else", nel codice mostrato viene analizzato un solo evento in quanto le operazioni sono simili per tutti gli altri, anzi nel caso specifico è uguale all'altro in quanto in quanto sono presenti solo due pulsanti che svolgono operazioni simili.

2.4 Architettura di rete

I due componenti base che permettono il funzionamento del sistema, ovvero il robot e l'applicazione, sono stati sufficientemente presentati. Vediamo ora qualche accenno sulla loro interconnessione.

2.4.1 Organizzazione della rete

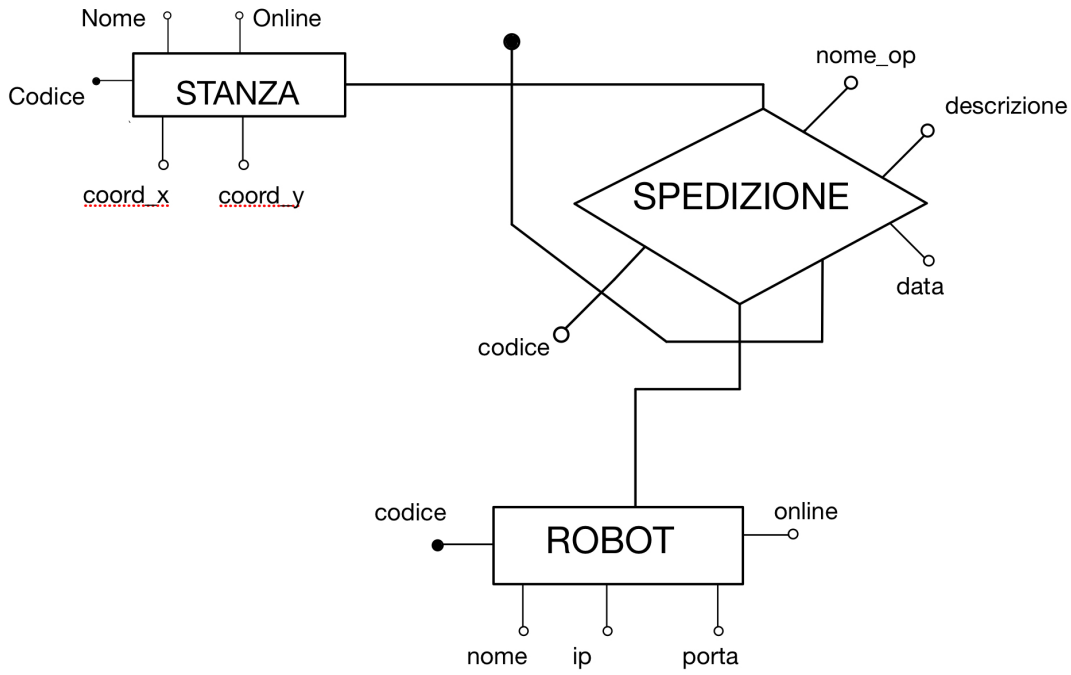
Tra i robot e l'applicazione c'è uno scambio di messaggi che rende necessario l'uso di una rete, in quanto si ipotizza che sia possibile usare l'applicazione per connettersi al server e mandare messaggi ai robot senza la necessità che essi siano connessi fisicamente tra di loro. Dunque, come visto nello schema ad inizio capitolo, una struttura da questo sistema è data da un server, n robot ed n client che possono accedervi. Questo può essere fatto attraverso l'uso di una local network, ovvero una struttura i cui i dispositivi appena citati sono connessi alla stessa rete. Si richiede inoltre una configurazione statica degli indirizzi ip assegnati al server ed ai robot, cioè una configurazione dove questi dispositivi abbiano sempre gli stessi indirizzi ip anche al loro spegnimento e riaccensione o al loro reset, anziché essere assegnati dal DHCP, un servizio ormai usato nelle reti moderne che permette un'assegnazione dinamica e automatica degli indirizzi. Questo, benché non obbligatorio, è necessario per un funzionamento corretto e più veloce, infatti l'applicazione è stata sviluppata in modo che alla sua apertura stabilisca una connessione al server attraverso un indirizzo noto a priori; inoltre la comunicazione con i robot avviene prelevando gli indirizzi ip memorizzati nel database ed associati ai nomi dei diversi robot; dunque se gli indirizzi dovessero cambiare continuamente, il database dovrebbe essere continuamente aggiornato e l'applicazione configurata con indirizzo e porta del server.
Per quanto riguarda la comunicazione, è stato detto che sia nei robot, sia nell'applicazione sono state implementate delle socket. Esse sono come delle prese (appunto dal loro nome che vuol dire presa in italiano) che collegano dispositivi; sono formate dall'associazione di un indirizzo ip ed una porta e possono essere di due tipi diversi a seconda del protocollo utilizzato. Posso essere socket TCP, ovvero un protocollo che necessita di stabilire una connessione prima di poter inviare messaggi, dunque in questo caso avviene uno scambio di messaggi iniziale per aprire la connessione, una volta aperta i dispositivi possono scambiarsi messaggi fino al chiudersi della connessione.

L'altro tipo è con l'utilizzo del protocollo UDP, esso necessita solamente dell'indirizzo e della porta su cui far viaggiare i pacchetti ed una volta spediti non si preoccupa se siano arrivati a destinazione o meno, il suo scopo è soltanto spedire i pacchetti in maniera facile e veloce senza chiedere all'altro dispositivo se vuole accettarlo, se è connesso, ecc.
Per questo sistema di spedizione è stato scelto di usare il protocollo UDP in quanto appunto più veloce e perché si tratta di spedire poche informazioni alla volta, sarebbe inutile generare un traffico dati riguardante richiesta di connessione e richiesta di chiusura da un punto di vista informatico.
Per quanto riguarda il server, potrebbe non essere un componente essenziale a seconda del caso di applicazione, infatti può benissimo essere aggiunta una sezione dell'applicazione in cui si chiede all'utente di inserire le informazioni di rete necessarie (come ip e porta) in modo da far avvenire una connessione diretta tra applicazione e robot senza passare per il database a chiedere informazioni. In questo caso è stato inserito un server che esegue un DBMS, ma se abbiamo detto che potrebbe anche non essere essenziale, si può analogamente pensare ad un server indispensabile che oltre al database esegue anche altri servizi, come un server HTTP per gestire le richieste di spedizione anziché gestirle con un'applicazione, oppure gestire una coda di richieste in caso esse siano tante, più di quelle che i robot presenti riescano a soddisfare.
Come detto in questo progetto viene eseguito il solo DBMS sul server, dunque daremo un'occhiata al database progettato per le prove simulate di questo progetto.

2.4.2 Il database

Un DBMS, DataBase Management System, è un software che consente la creazione, la manipolazione e l'interrogazione di un database ed è in genere eseguito su hardware dedicato. Ne esistono di diversi, ma quello utilizzato in questo progetto è MySQL, DBMS relazionale, ovvero che si basa sul modello relazionale dei dati il quale è anche quello oggi più diffuso. Inoltre permette l'accesso ai dati da software esterno attraverso API disponibili per diversi linguaggi, tra cui il JDBC che è quello utilizzato.
Anche se il sistema è stato pensato in modo generico, ovvero può essere adattato a diversi usi grazie alla scalabilità di robot e applicazione, un database non può essere riadattato facilmente, quindi a seconda della casistica è richiesta la progettazione di un database adeguato allo scopo.
In questo progetto ne è stato realizzato uno minimale per studiare e capire il funzionamento del sistema in generale.

Si suppone che ci siano n robot ed m stanze, ogni robot può servire qualunque stanza ed ogni stanza può essere servita da qualunque robot. Inoltre si deve tenere traccia di tutte le spedizioni effettuate.
Uno schema UML che rappresenta questa base di dati può essere:



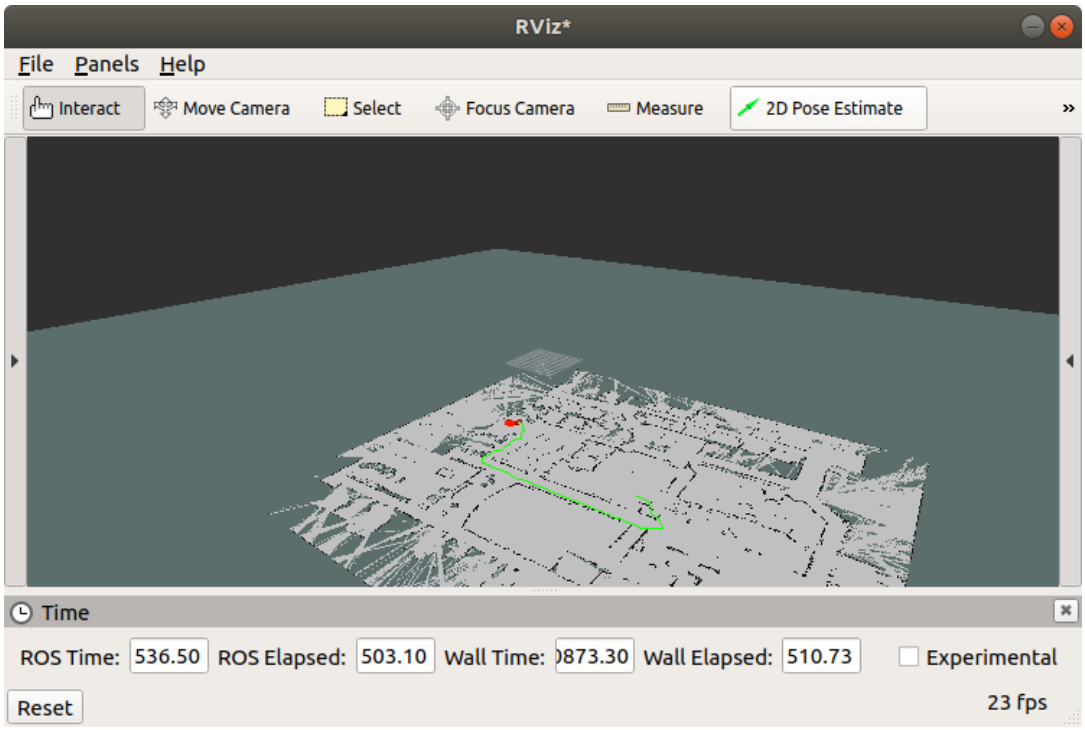
Questo diagramma dà origine ad un database composto da tre tabelle: stanza, robot e spedizione.
Stanza è identificata dal codice e ad essa è associato un nome, le coordinate in cui si trova e se in quel momento è online, ovvero è un luogo che può essere servito; robot è identificato da un codice e anch'esso ha associato un nome ed è di interesse sapere se è online, cioè se può effettuare un servizio di spedizione in quel momento, inoltre ci sono le informazioni essenziali di ip e porta alle quali l'applicazione può contattare il robot stesso; spedizione è identificata da un codice, dal robot che ha effettuato il servizio e dalla stanza servita, ha inoltre informazioni inerenti il nome di chi ha effettuato la spedizione, la descrizione in cosa consisteva e la data.

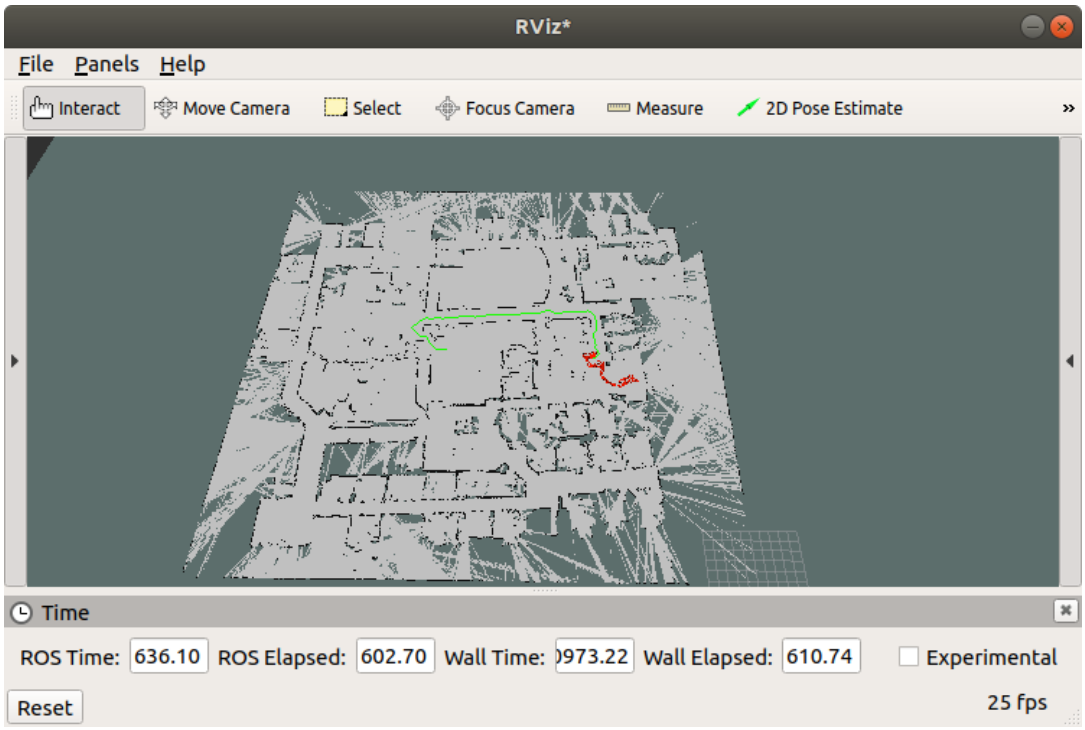
Capitolo 3:
Impiego del sistema automatizzato

Nei precedenti capitoli si è affrontato l'argomento del progetto in un ambito molto tecnico, descrivendo strumenti e funzionalità di tutto il materiale usato durante la progettazione e lo sviluppo di questo progetto. Come detto più volte è stato pensato in un modo molto scalabile che permette l'idea applicabile a diversi contesti, un pò come un software open source che può essere adottato e migliorato. Proprio per questa caratteristica, questo capitolo si pone l'obiettivo di esplorare vari casi di applicazione e di fare alcune ipotesi che non sono state fatte nel progetto in questione; ma questo non prova di aver visto un caso di test virtuale.

3.1 Test ed esempi di applicazione

Il test, come tutto il progetto è avvenuto in un ambiente di simulazione, vediamo di seguito i passi fatti e con un'immagine vediamo i risultati.
Attraverso una macchina virtuale con Ubuntu a bordo, sono stati eseguiti i nodi ros per simulare il robot, in particolare sono stati lanciati da linea di comando il nucleo di ros, l'ambiente stage, la mappa con map_server, thin_localizer, thin_planner ed il nodo progettato per gli obiettivi, il quale si è messo in attesa di ricevere informazioni sulla socket creata. Questi processi in esecuzione su un robot sarebbero configurati in modo da partire in autonomia all'accensione del robot. Il server è eseguito su un secondo calciatore connesso alla rete locale con ip già impostato e configurato nell'applicazione eseguita su un terzo dispositivo anch'esso connesso alla rete locale. Ovviamente è stato lanciato anche rviz per visualizzare i risultati, come si può osservare dalle immagini a pagina seguente.
Quando ros riceve i messaggi e spedisce un goal al proprio Simple Action Server, dapprima si vede un intorno di punti rossi che rappresentano l'odometria del robot, successivamente viene tracciato un percorso verde tra il robot ed il punto indicato come obiettivo in modo giusto grazie all' opera del global planner viene; nella seconda immagine si può osservare il robot in movimento, il quale non esegue il percorso perfettamente, tuttavia segue la direzione giusta grazie anche al local planner che ricalcola il percorso nelle vicinanze del robot.


RViz*
File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
Time
ROS Time: 536.50
ROS Elapsed: 503.10
Wall Time: 873.30
Wall Elapsed: 510.73
Experimental
Reset
23 fps


RViz*
File Panels Help
Interact
Move Camera
Select
Focus Camera
Measure
2D Pose Estimate
Time
ROS Time: 636.10
ROS Elapsed: 602.70
Wall Time: 973.22
Wall Elapsed: 610.74
Experimental
Reset
25 fps

Approfondiamo ora alcuni casi di utilizzo.
Un primo caso può banalmente essere quello rappresentato nella simulazione: come si evince dal database, si fa riferimento al caso in cui ci sono un tot di stanze, quindi ci si può trovare ad esempio in una grande azienda con molti uffici (preferibilmente ad un solo piano a meno di non utilizzare un agente robotico in grado di fare le scale e/o prendere l'ascensore) dove i dipendenti di diverse stanze hanno il bisogno di scambiarsi spesso documenti di vario genere, in questo caso grazie alle due diverse modalità di spedizione un impiegato può far venire il robot nel proprio ufficio e consegnarli la merce oppure andare nella postazione centrale dei robot ed incaricane uno.
Pensiamo al caso di un ospedale, dove ad esempio occorre portare un medicinale in una stanza ma tutti gli infermieri sono impegnati oppure, caso ancora più realistico, un reparto di malattie infettive dove vi si può accedere solo con tute di massima sicurezza (e dove l'infetto è in grado di prendere i medicinali in modo autonomo). Ecco in questo caso, con un'opportuna mappatura della struttura, più robot potrebbero essere di aiuto nel trasporto di medicinali o altro.
Altro caso, ambiente scolastico. Girano spesso in una scuola circolari riguardanti importanti informazioni per studenti ed insegnati senza scomodare un collaboratore magari attualmente impegnato nella sorveglianza, nonché c'è sempre una maestra che chiede ad un alunno di andare a prendere il gesso per la lavagna o altri strumenti. In questo caso potrebbero essere utili dei robot opportunamente modificati, ad esempio insieme alla richiesta di spedizione viene richiesto anche di cosa si necessita, così un collaboratore scolastico può fornire il materiale al robot che effettuerà la consegna.
Pensiamo al caso di un albergo con molte camere, nel caso in cui gli ospiti chiedano qualcosa, oppure deve essere consegnato il menu del giorno ogni giorno per permettere alla struttura di organizzare i pasti, in questo caso un robot può fare il giro delle stanze magari emettendo un segnale acustico o visivo all'interno della stanza per annunciare il suo arrivo in modo che gli venga aperta la porta. Solitamente le strutture alberghiere hanno più piani, si può pensare ad un robot per piano.
Consideriamo un centro di raccolta rifiuti automatizzato, in questo caso si può fare una modifica all'applicazione in modo da poter inserire il tipo di rifiuto da buttare, in questo modo se ci si trova in un ambiente molto grande dove i diversi rifiuti sono raccolti in diversi punti, il robot potrà guidarti correttamente verso il punto giusto.

3.2 Oltre il limite: ipotesi aggiuntive

Nonostante il progetto è stato svolto in ambiente di simulazione, il tutto è facilmente applicabile al caso reale se non fosse per un componente molto delicato da comprare e configurare, ovvero il laser, insieme alla mappatura dell'ambiente circostante. Per il resto scelti motori, struttura sensori banali come i rilevatori di ultrasuoni e grazie ad una semplice scheda come può essere una Raspberry Pi, il gioco è più che fatto.
Visto dunque che il laser è il primo componente sul quale si è fatta l'ipotesi di averlo, si possono fare ipotesi aggiuntive ed ampliare le funzionalità di un robot nel lungo periodo.
Si può implementare un nodo che permetta ai robot di comunicare tra di loro, questo può aiutare a scambiarsi messaggi e se ad esempio c'è un robot con più richieste mentre un altro è libero, quello occupato può incaricare direttamente quello libero senza rifiutare il servizio.
Facendo un passo avanti si può dotare il robot di bracci e pinze in grado di afferrare oggetti, in questo caso esso diventa più autonomo nella casistiche prima elencate e può evitare l'ausilio di un essere umano finora dato per scontato per caricare la spedizione. Ovviamente entrano in gioco anche fattori come il riconoscimento di immagini.
In un mondo sempre più connesso si può far interagire il robot con l'IoT, ad esempio comandando il robot attraverso comandi vocali, usando agenti intelligenti come Amazon Alexa, Google Home, Siri.

Conclusioni

Si possono dichiarare raggiunti gli obiettivi dati dall'idea alla base di questo progetto.
E' stato sviluppato un nodo ros in grado di mettersi in ascolto su una socket e , dopo aver ricevuto i dati, in grado di creare un obiettivo, detto goal e raggiungerlo, in particolare l'obiettivo per ogni robot di questo progetto è raggiungere un punto destinazione. Si è visto che il percorso viene calcolato in modo ottimale grazie al modulo del global planner che usa algoritmi per trovare i cammini minimi, come l'algoritmo di Dijkstra, mentre è da migliorare il lavoro del local planner per permettere al robot di seguire al meglio la traiettoria, mentre è stato notato che a il robot subisce intoppi lungo il percorso, ad esempio può non riuscire a schivare ostacoli istantanei oppure può perdere l'orientamento.
E' stata sviluppata anche una piattaforma con una discreta interfaccia che si collega ad un server per raccogliere le informazioni che sono necessarie alla comunicazione con il robot, è stata sviluppata il più scalabile possibile in modo che possa essere adattata a diversi scopi.
Sono stati risolti problemi di connessione tra app e server e tra app e robot assegnando indirizzi ip statici in una rete locale.

Ringraziamenti

Se me lo avessero detto da piccolo non ci avrei mai creduto. Ho sempre visto gente che frequentava l'università solo nei film ed a volte pensavo fosse una leggenda, oppure quando ne sentivo parlare pensavo fosse solo per gente veramente brava e con tanta passione.
Dopo la scuola secondaria di primo grado, l'obiettivo era terminare gli studi quanto prima magari senza debiti in modo da godersi le vacanze estive, ma iniziato il percorso alle scuole superiori qualcosa è iniziato a cambiare, quella passione che prima pensavo fosse un lusso solo per alcuni, ho cominciato a capire che possiamo averla tutti. Così ho cominciato a pensare che l'università non era poi così fuori portata per me, anche perché crescendo con i giusti insegnamenti gli ostacoli si impara a superarli, da piccoli e senza esperienza sembra sempre tutto troppo grande, ma infine esame dopo esame, tra gioie e dolori, riesco a concludere un ciclo triennale con grande meraviglia, entusiasmo, gioia.
Un grazie speciale a tutti gli insegnati che in questi anni mi hanno portato sulla strada giusta fin dalle scuole elementari, dunque tutte le maestre, tutti i professori delle scuole medie e tutti i professori di cinque fantastici anni di istituto tecnico. Ma anche i professori universitario che nella maggior parte dei casi riescono a trasmettere quella passione che un giorno probabilmente diventerà un lavoro, anche perché come diceva Steve Jobs, se si ha passione non si lavorerà un solo giorno della propria vita.
Un grazie speciale ovviamente ai miei genitori e mia sorella che mi hanno sostenuto e permesso di studiare sin dal primo giorno di questa avventura, e poi un grazie al resto della mia grande e meravigliosa famiglia, compresi tutti gli amici che hanno condiviso con me gioie e dolori in questi anni.
Ma il mio percorso non si ferma, continua ed è anche merito di tutti voi che ho citato, dunque, ancora grazie.

Bibliografia

- http://wiki.ros.org
- https://www.hindawi.com/journals/jat/2018/6392697/
- Richard Vaughan. "Massively Multiple Robot Simulations in Stage", Swarm Intelligence 2(2-4):189-208, 2008.
- Brian Gerkey, Richard T. Vaughan and Andrew Howard. "The Player/ Stage Project: Tools for Multi-Robot and Distributed Sensor Systems" Proceedings of the 11th International Conference on Advanced Robotics, pages 317-323, Coimbra, Portugal, June 2003