ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

LA PIATTAFORMA ROS PER LO SVILUPPO DI APPLICAZIONI PER LA ROBOTICA: PANORAMICA E CASO DI STUDIO

Elaborato in
PROGRAMMAZIONE DI SISTEMI EMBEDDED

Relatore
Prof. ALESSANDRO RICCI

Presentata da
LORENZO CROCCOLINO

Anno Accademico 2018 – 2019

Ai miei genitori
a mia sorella Eleonora
ad Alessia

Indice

Introduzione

La robotica negli ultimi anni ha raggiunto sempre di più il successo soprattutto grazie al largo impiego all'interno dell'industria manifatturiera. I robot più utilizzati in questo momento sono sicuramente i cosiddetti bracci meccanici che operano all'interno delle linee di montaggio sostituendo l'uomo in processi faticosi e ripetitivi.

Pur essendo molto importanti e rappresentando un grosso passo in avanti rispetto a quando qualche anno fa la forza lavoro era esclusivamente affidata all'uomo, negli ultimi anni si è sentito il bisogno di qualcosa che aiutasse l'operaio non solo nello svolgimento di lavori statici e ripetitivi ma anche all'interno di mansioni più complesse ed ambiti più dinamici.

La robotica di servizio è proprio questo, ovvero, lo sviluppo di robot che affiancano l'uomo durante i processi produttivi in modo da alleggerire il carico di lavoro e aumentare la produttività in relazione alle ore di lavoro. Questa nuova tipologia di automi ha permesso alla robotica di approdare in qualsiasi settore, da quello medico a quello agricolo, facendo affluire al suo interno differenti discipline scientifiche e umanistiche. La meccanica, l'informatica, le scienze sociali e la medicina sono solo una minima parte.

Caratteristiche essenziali per un robot di servizio sono la capacità di rilevare le entità all'interno dell'ambiente che lo circonda e muoversi in modo autonomo a servizio dell'uomo senza recare danni a cose e persone. Per fare ciò negli anni sono stati sviluppati algoritmi che permettono la navigazione all'interno di ambienti sconosciuti ed in totale sicurezza, con la possibilità di creare mappe in tempo reale coerenti con l'ambiente che circonda il robot.

Per aiutare gli sviluppatori e rendere la programmazione dei robot sempre più efficiente sono stati rilasciati diversi framework che hanno ricevuto un grande appoggio sia dalle community di programmatori che da grandi soft-

ware house. Alcuni dei più utilizzati oggi sono Microsoft Robotics Developer Studio, Mobile Robot Programming Toolkit (MRPT) e Robotic Operating System (ROS). Durante questa tesi si andrà ad approfondire l'architettura, il funzionamento e lo sviluppo proprio di quest'ultimo, ROS.

Oltre ad una rassegna dei componenti software, delle funzionalità e delle caratteristiche peculiari che fanno di questo software uno dei più diffusi all'interno dell'industria robotica, verranno illustrati due casi di studio inerenti proprio alla robotica di servizio e sviluppati attraverso l'impiego di ROS. Il primo prevede un robot in grado di esplorare un ambiente simulato e simultaneamente mapparlo. La navigazione è affidata ad un algoritmo che rende questa operazione autonoma oppure ad un operatore che è in grado, tramite un'interfaccia grafica, di comandare in modo puntuale il robot. All'interno del secondo caso invece è proposta un'esperienza reale dove viene descritto lo sviluppo di un robot per il trasporto di merci con funzione di inseguimento dell'operatore. All'interno dello studio viene descritto l'iter di sviluppo a partire da una prima versione con tecnologie molto semplici fino ad arrivare, tramite evoluzioni implementative, ad una versione più complessa e all'adozione di ROS.

Capitolo 1

Piattaforma ROS

1.1 La robotica di servizio

Secondo la Federazione Internazionale di Robotica (IFR) un robot di servizio è "un robot che opera in maniera autonoma o semi-autonoma per compiere servizi utili al benessere degli esseri umani, escludendo l'ambito manifatturiero"[1] . Questa è la definizione che descrive i robot più recenti e più intelligenti che negli ultimi anni si sono fatti strada nella ricerca e nel mercato. Dalla logistica all'agricoltura, ogni settore ha negli ultimi anni iniziato la sua convivenza con i robot.

Uno dei settori più attivo, se non il più attivo, per questa tipologia di automazione è sicuramente quello della logistica, con un incremento di vendite di robot intelligenti del 25-30% ogni anno, basti pensare che nel 2017 il 63% dei robot di servizio per uso professionale era di tipo logistico. Da allora, la loro popolarità è cresciuta in modo costante, al punto che l'IFR prevede che le vendite saliranno a 600.000 unità entro il 2021.

La crescente aspettativa dei clienti riferita a questa tipologia di servizi e la carenza di una vera e propria offerta, hanno favorito fortemente la ricerca. Sono nate in questi anni diverse soluzioni interessanti che velocemente sono entrate a far parte del processo produttivo, non solo delle aziende già leader in questo settore, come Amazon per citare la più famosa, ma anche in quelle medie e piccole. L'uso combinato e rivoluzionario di bracci meccanici e robotica mobile ha permesso inoltre di ottimizzare in modo sensibile le risorse su tutta

la filiera, anche con budget relativamente ridotti, migliorando sensibilmente i processi aziendali in modo molto trasversale, economicamente parlando.

La robotica di servizio ad oggi rappresenta una realtà e non più solamente un argomento relegato alla ricerca o ad un pensiero futuristico, essa rappresenta già il presente e lo fa da diversi anni ormai, in particolar modo all'interno del settore B2B (business-to-business). L'agricoltura e la sanità sono altri due settori che insieme alla logistica hanno beneficiato di questa nuova spinta tecnologica e che riescono a segnare ogni anno incrementi a doppia cifra (20-25%, stima l'IFR) nelle vendite di robot che collaborano attivamente con gli esseri umani. Ricerche riferiscono che questa crescita manterrà il passo per almeno altri 4 anni prima di subire variazioni.

Esoscheletri per i lavori pesanti e ripetitivi, come quelli della catena di montaggio auto, sono un'altra fra le tantissime facce della robotica di servizio e dei robot collaborativi. Fino a poco tempo fa tecnologie come queste venivano considerate realtà lontane ed irrealizzabili nel breve periodo ma che oggi sono adottate dalle più grandi industrie nel mondo che hanno creduto ed investito nelle tecnologie ma che presto rappresenteranno uno standard anche in quelle più piccole.

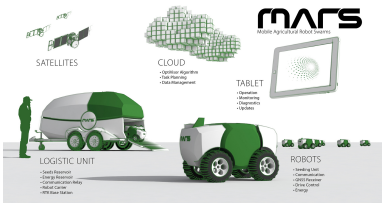
Figura 1.1: Fotografia che ritrae due operai che indossano un esoscheletro all'interno di una catena di montaggio per la fabbricazione di auto del gruppo BMW [2]

Ritornando invece a parlare di agricoltura, l'utilizzo di piccoli robot autoguidati per la semina, come per esempio il progetto MARS (Mobile Agricultu-

ral Robot Swarms) che è possibile vedere nell'immagine, riusciranno a ridurre i costi energetici, le emissioni di polveri sottili, l'ingombro a terra e ad ottimizzare l'efficienza di operazioni che solitamente impiegano trattori e macchine dal forte impatto ambientale[3].

Lo stesso discorso vale anche per droni aerei, spesso utilizzati come robot di servizio per il monitoraggio delle colture oppure per la distribuzione di insetti utili e diserbanti, compiti che fino ad oggi erano affidati a voli di piccoli aerei e quindi applicabili solamente a coltivazioni di grandi dimensioni ma che in ogni caso risultavano scomodi e non certo ottimizzati. Tutto ciò si traduce in maggiore tempestività nell'intervenire contro una determinata malattia, ottimizzazione dei costi di manutenzione e relativo aumento della quantità e della qualità della produzione.

(a) Fotografia aerea che mostra numerosi robot MARS al lavoro in modo coordinato e simultaneo [4]



(b) Immagine che mostra in che modo i robot MARS lavorano e ricevono informazioni [4]

Figura 1.2: Immagini relative al robot di servizio MARS

Per quanto sia ancora in proporzioni ridotte rispetto all'ambito professionale, anche quello domestico possiede la sua fetta di utenti. Ospitare in casa robot che ogni giorno svolgono in modo autonomo lavori domestici non è più una novità sensazionale e, anche se ancora rappresentano solamente una piccola parte dei robot venduti, sempre l'IFR stima che nelle case entreranno oltre 42 milioni di nuovi robot prima della fine del 2020, una cifra davvero importante se la si compara ai numeri del 2018 che ha visto una crescita di "soli" 16 milioni di unità vendute.

Figura 1.3: Fotografia di un robot aspirapolvere al lavoro in ambito domestico

1.2 Piattaforma ROS

ROS (Robotic Operating System) è un software middleware utilizzato per lo sviluppo di applicazioni per la robotica, in particolare per la programmazione di robot di servizio.

Nato nel 2007 dai laboratori della Stanford University (Stanford Artificial Intelligence Laboratory), per quanto il nome possa trarre in inganno, ROS non è un vero e proprio sistema operativo. Prima di tutto infatti è un framework open-source che mette a disposizione dello sviluppatore strumenti e librerie per la programmazione di applicazioni robotiche a partire dalla loro scrittura fino ad arrivare al momento del debug. Allo stesso tempo presenta alcune funzioni simili a quelle di un sistema operativo classico come, per esempio gestione dei processi, dei pacchetti, intesi come parti del software, e delle loro dipendenze. Le funzioni di middleware vengono invece sfruttate per la comunicazione tra processi e macchine differenti.

ROS è pensato per operare in simbiosi con sistemi operativi Linux, in particolare i principali OS su cui viene portato avanti lo sviluppo sono Ubuntu e Debian. Da poco sono entrati a far parte di una fase sperimentale anche versioni per Gentoo, Arch Linux , Fedora ed altre distribuzioni Linux, ma ancora non è stata sviluppata nessun tipo di versione per Windows. La compatibilità di ROS è continuamente in espansione grazie ai numerosissimi pacchetti che ne ampliano la portabilità, spesso sviluppati e supportati dalla community

stessa. Un esempio fra tutti è proprio una versione di ROS non stabile su dispositivi Android, sviluppata e supportata interamente dalla community in forma sperimentale. All'interno del middleware i pacchetti che lo compongono sono innumerevoli e costantemente in crescita, questo garantisce completa compatibilità ed affidabilità con le sempre nuove tecnologie dell'industria.

ROS infine non costituisce un linguaggio di programmazione, ma integra ufficialmente codice scritto in C++, Python e Lisp. Esistono delle librerie sperimentali per l'integrazione e lo sviluppo di codice in Java e Lua, anch'esse supportate a livello di community.

1.3 Benefici e vantaggi

ROS ha come obiettivo quello di differenziarsi dai classici framework pensati per la robotica, non tanto per un parco di funzioni maggiore, ma perché punta tantissimo sulla facilità e sulla rapidità dello sviluppo. Questo è reso possibile dal suo design fortemente modulare che si presta benissimo al riuso del codice sia in attività di ricerca che di sviluppo. I pacchetti che compongono il software sono fortemente riutilizzabili e possono facilmente essere condivisi in altri progetti attraverso un banale copia e incolla, a patto di rispettarne le dipendenze. Questa sua peculiare architettura è in grado di apportare grandi vantaggi in merito alla gestione del progetto ed alla comunicazione interna del team di sviluppo, a cui viene offerta grande indipendenza ed allo stesso tempo un ottimo livello di collaborazione.

ROS è pensato inoltre per essere utilizzato all'interno di sistemi distribuiti, il che si sposa davvero bene con la logica modulare descritta sopra applicata al mondo della robotica. È sicuramente un ulteriore grande vantaggio quello di poter sviluppare software per ROS utilizzando linguaggi di programmazione moderni e largamente diffusi come C++, Python e Lisp, rendendo l'approccio iniziale più semplice e meno "traumatico", ampliando in questo modo anche il bacino di potenziali nuovi sviluppatori interessati ad utilizzare il middleware.

ROS infine presenta un'ulteriore facilitazione per i programmatori. Spesso infatti la ricerca e la risoluzione degli errori è un processo complesso, soprattutto in ambito robotico dove l'hardware e la fisica mettono i bastoni fra le ruote a chi cerca di eseguire test ed aggiornamenti software: ROStest è uno

unit/integration test framework sviluppato su misura per eseguire unit-test tra i molteplici nodi del software e che risolve a pieno il problema descritto.

1.4 Esempi di utilizzo

Come detto in precedenza, ROS è una delle soluzioni software open-source più diffusa all'interno del mondo della robotica. Questa fama, del tutto meritata, ha permesso al middleware di essere utilizzato non solo all'interno di progetti amatoriali e da piccoli team di sviluppo indipendenti ma anche e soprattutto da aziende leader nel proprio settore.

1.4.1 NASA

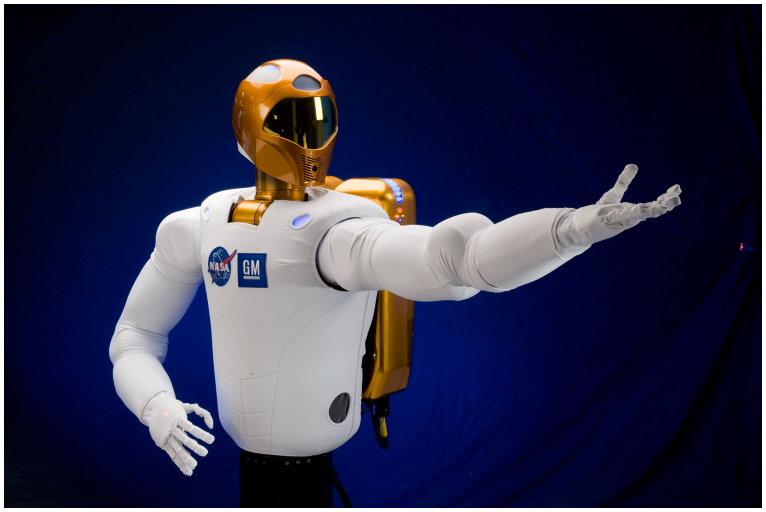


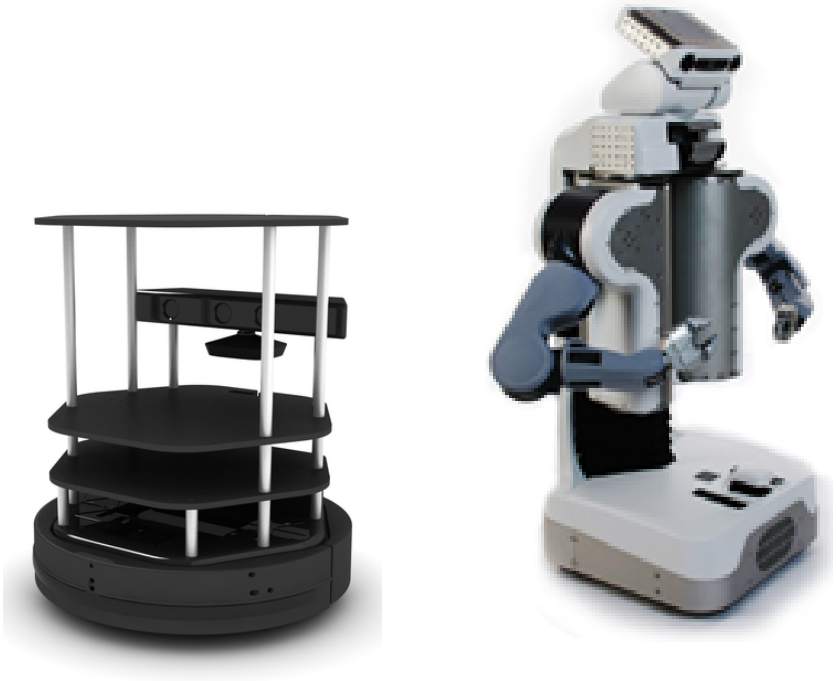
Figura 1.4: Fotografia che immortala la nuova versione dell'umanoide Robonaut il giorno della presentazione [5]

Il primo esempio di utilizzo per importanza non può che essere la NASA. L'agenzia americana governativa, che si occupa della ricerca e sviluppo di soluzioni in grado di lavorare in ambienti ai limiti della sopravvivenza come quelli dello spazio, ha scelto nel 2010 di presentare il suo Robonaut 2 (R2) con a bordo ROS [6]. Come si legge all'interno della documentazione, il passaggio dalla prima versione alla seconda cambiò radicalmente le performance dell'umanoide: grazie all'adozione del software ROS questa versione ottenne maggiore

destrezza nei movimenti, maggiore affidabilità ed uno sfruttamento migliorato dei sensori a bordo. Il robot fu pensato per affiancare spalla-a-spalla gli astronauti all'interno della stazione spaziale durante le varie fasi degli esperimenti, all'esterno della stazione per interventi di manutenzione e per l'esplorazione di pianeti e satelliti. Proprio la Luna fu scelta come primo grande traguardo del robot dopo un periodo di ottimi risultati ottenuti all'interno della stazione spaziale, traguardo sfumato con l'assorbimento del robot all'interno di un nuovo progetto ancora in corso chiamato "Project Morpheus".

Robonaut 2 fu il primo robot umanoide ad essere lanciato nello spazio.

1.4.2 Willow Garage



(a) Immagine del robot Turtlebot [7]

(b) Immagine del robot PR2

Figura 1.5: Due dei robot più conosciuti sviluppati da Willow Garage

Willow Garage è una società che si occupa principalmente di ricerca e sviluppo hardware e software open-source in ambito robotico. Questa società è stata fra i principali responsabili della nascita di ROS. Utilizzando proprio il famoso middleware ha progettato, sviluppato e venduto diversi modelli di cosiddetti "personal robot". Questi robot sono in grado di vivere fianco a fianco alle persone e di aiutarle all'interno di ambienti domestici e professionali in compiti che vanno dal trasporto di oggetti alla telepresenza, passando per umanoidi in grado di svolgere attività simili a quelle di un maggiordomo o di una segretaria. Il modello più conosciuto ed uno dei primi modelli funzionanti e venduti fu TurtleBot. Venduto principalmente come base per lo sviluppo, è equipaggiato con sensori di riconoscimento visuale ed è in grado di muoversi a 360 gradi con un piccolo piano superiore per il trasporto degli oggetti. Sicuramente un ottimo progetto fu anche l'ultimo in ordine cronologico, ovvero il robot PR2 (Personal Robot 2), robot dalle fattezze umanoidi costruito anch'esso come piattaforma per lo sviluppo di applicazioni robotiche orientate all'uso domestico e business.

1.4.3 Pilz

Figura 1.6: Immagine rappresentativa dei due principali modelli di robot open-source venduti da Pilz S.p.a. [8]

Pilz, azienda italiana leader nel settore dell'automatizzazione con più di 2500 dipendenti e più di 40 sedi sparse in tutto il mondo, si occupa di sviluppare soluzioni verticali per un'automazione sicura a servizio dell'uomo. I

prodotti progettati, sviluppati e venduti dall'azienda sono diversi e vanno dai sensori ai sistemi di controllo passando per robot e attuatori, tutti con una peculiare attenzione alla sicurezza dell'uomo con cui i sistemi si troveranno a collaborare. Si tratta quindi di un'azienda incentrata sulla produzione di apparati più o meno grandi per il mondo della robotica di servizio. La parte però interessante che differenzia Pilz da molte altre aziende è il fatto che sia possibile acquistare quelli che loro definiscono "Moduli ROS" [9]. Questi moduli non sono altro che prodotti, come bracci meccanici o robot su ruote, totalmente open-source ed equipaggiati con ROS. In questo modo è possibile personalizzare e far calzare a pennello il nuovo hardware in una catena di montaggio già esistente e personalizzata oppure costruirne una nuova su misura. In ogni caso è possibile in questo modo utilizzare il team di programmatori interno all'azienda sia nella parte di personalizzazione che in quella di manutenzione, rimanendo indipendenti e assicurando così un supporto life-time.

1.5 Nel prossimo capitolo

Come visto qui sopra, ROS è un sistema largamente utilizzato sia da piccoli team che da grandi aziende. Questo grande successo non può che essere dovuto anche ad un'architettura estremamente flessibile e ben progettata.

All'interno del prossimo capitolo verrà discussa proprio l'architettura di un'applicazione sviluppata su ROS, mettendo in evidenza come questa è organizzata e in che modo è in grado di funzionare e di sfruttare le API messe a disposizione dal middleware.

Capitolo 2

Architettura di un'applicazione ROS

2.1 Introduzione

ROS è un middleware dall'architettura fortemente ispirata a quella delle reti di telecomunicazioni di tipo TCP/IP.

La struttura generale di un'applicazione si può immaginare infatti come una rete di nodi, i quali, interconnessi fra loro, scambiano messaggi all'interno di appositi bus ed accedono a servizi messi a disposizioni da altri nodi.

Si può accostare la figura dei nodi su ROS a quella dei client connessi ad una rete, dove ognuno di questi accede a servizi hostati su altri client, che saranno quindi dei server, e scambia messaggi con altri client all'interno della medesima rete.

All'interno di questo capitolo verranno esplorati i principali componenti che costituiscono lo scheletro sul quale un'applicazione sviluppata utilizzando ROS si basa.

2.2 Master

Master è un nodo unico all'interno dell'architettura di ROS che si occupa di assegnare un nome e registrare ogni singolo nodo connesso al sistema come publisher, subscriber o service provider.

Per utilizzare nuovamente il paragone con le reti TCP/IP, esso ha un comportamento vagamente simile a quello del server DHCP che si occupa di registrare i client e distribuire indirizzi IP in una classica rete internet.

Al master viene assegnato un well-known XML-RPC URI in modo che qualsiasi nodo creato sia sempre in grado di comunicare con esso. Sono infatti i singoli nodi a contattare il master nel momento in cui hanno la necessità di eseguire subscribe o publish a/su determinati topic. Sarà il master a fornire una lista di nodi ed a negoziare il tipo di connessione che utilizzeranno i due per comunicare, in modo tale che poi siano loro stessi a farlo in modo diretto attraverso una connessione di tipo peer-to-peer.

Le operazioni appena descritte sono rese possibili mediante lo sfruttamento di API basate su protocollo XML-RPC-stateless.

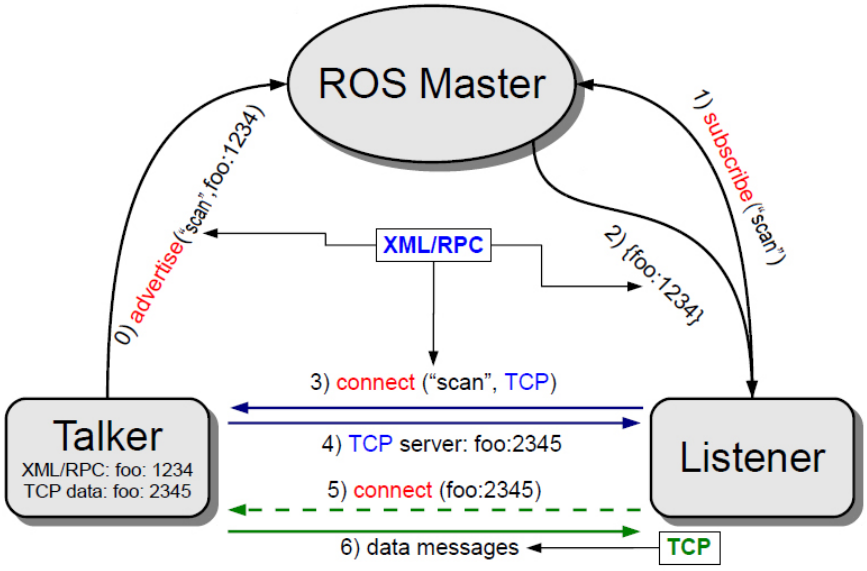


Figura 2.1: Esempio di come vengono sfruttate le API del master dagli altri nodi [10]

La figura qui sopra è decisamente esemplificativa di quello descritto prima. Si nota con estrema semplicità come il nodo nominato come "Talker", rappresentante un generico *publisher*, contatta il master notificando attraverso il metodo *advertise* il nome del topic su cui andrà a pubblicare i messaggi e di seguito il proprio URI. Anche il secondo nodo chiamato "Listener", un generico

subscriber, contatta il master notificando a sua volta il nome del topic su cui è intenzionato a rimanere in ascolto. Il master a questo punto fornisce al secondo nodo i parametri per entrare in contatto con il primo nodo, in particolare l'URI del *publisher*, sono poi loro stessi a finalizzare la connessione.

2.3 Parameter server

Parameter Server è una componente che fa parte parte del master e mette a disposizione, proprio come il primo, delle API basate su protocollo XMLRPC-stateless. Grazie a queste API è possibile storicizzare e rendere pubblici parametri statici.

Questi valori sono utilizzati come parametri di configurazione durante la fase di runtime dai nodi ed è possibile visualizzare e modificare il valore di ogni singolo parametro utilizzando i tool messi a disposizione da ROS, come per esempio *rosparam*.

Pur riducendo in certi casi le prestazioni del software, sono una funzionalità molto apprezzata dagli sviluppatori che, durante la fase di debugging, possono testare in modo più dinamico configurazioni differenti direttamente durante l'esecuzione del software.

I tipi di dato che è possibile utilizzare per ogni parametro sono:

- integer a 32-bit
- booleani
- stringhe
- double
- date secondo lo standard iso8601
- liste
- base64-encoded binary data

2.4 Nodes

I nodi, o *nodes*, sono processi che si occupano dell'effettiva computazione dei dati e che svolgono le principali funzioni all'interno del sistema. Tutti i nodi vengono inseriti all'interno di un grafo interconnesso dove ognuno di essi è in grado di comunicare con tutti gli altri in modo diretto.

Ogni nodo ha sempre accesso alla comunicazione con il nodo master. Un nodo di norma svolge poche e precise funzionalità, viene scoraggiata l'implementazione di nodi onnipotenti che svolgono troppe funzioni, in questo modo viene mantenuto ordinato ed intuitivo il grafo.

Le API che ogni nodo mette a disposizione di default sono :

- *slave API*, sono XML-RPC API che permettono ad ogni nodo di comunicare con il nodo *master* e la negoziazione del tipo di connessione da utilizzare per lo scambio di messaggi con un secondo nodo

- *topic transport protocol implementation*, permettono ai nodi di stabilire connessioni dirette fra loro e di utilizzare protocolli di rete TCP ed UDP per lo scambio di messaggi

- *command-line API*, permettono la configurazione dei nomi dei nodi nel momento dell'esecuzione

E' possibile ottenere una lista dei nodi ed eseguire operazioni sugli stessi utilizzando il tool nativo messo a disposizione da ROS *rosnode*.

I nodi sono solitamente sviluppati in C++, Python e Lisp, che sono i linguaggi ufficialmente supportati da ROS per lo sviluppo. E' possibile utilizzare altri linguaggi di programmazione utilizzando librerie aggiuntive in versione sperimentale.

2.5 Instradamento dei messaggi

Per lo scambio dei messaggi fra i nodi ROS utilizza principalmente due protocolli di rete, UDP e TCP a seconda del tipo di connessione utilizzata. Il protocollo TCP viene solitamente utilizzato in caso di connessioni Ethernet. Risulta più sicuro rispetto al secondo grazie a meccanismi di controllo sul

messaggio inviato che ne garantiscono una ricezione ordinata e una ritrasmissione in caso di errori. Nel caso in cui però la trasmissione dati sia affidata a reti WiFi o cellulari, le caratteristiche di solidità e sicurezza del protocollo TCP si traducono in scarse performance e perdita di informazioni. Proprio per questo motivo, in casi come questo, viene preferita una connessione UDP. Non prevedendo meccanismi di ritrasmissione e garanzie sull'ordine di arrivo, infatti, questo protocollo risulta molto più performante e paradossalmente con una perdita di informazioni minore rispetto al primo.

La scelta del tipo di protocollo viene effettuata dai singoli nodi prima della trasmissione dei dati, attraverso lo sfruttamento di apposite API, come descritto nelle sezioni precedenti.

Capitolo 3

Sviluppare su ROS

3.1 Livelli concettuali

Per sviluppare sistemi basati su ROS è necessario comprendere i principali livelli concettuali su cui il middleware è basato, ovvero:

- File system level
- Computation graph level
- Community level

I livelli concettuali, dalla wiki "Concepts", permettono di organizzare e gestire al meglio ogni progetto. Essi indicano in modo dettagliato come implementare la struttura del file system, la gestione delle comunicazioni all'interno dell'applicativo, come eseguire in modo accurato debug ed analisi dei dati, come interfacciarsi con la community durante tutta la fase di sviluppo e molto altro ancora.

3.1.1 File System Level

Il livello del File System comprende tutte quelle risorse che sono salvate fisicamente su disco, in particolare:

- Packages
- Metapackages

- Manifest
- Message types
- Service types

Packages

I package sono la struttura principale per l'organizzazione del software ROS [11]. Essi contengono processi, librerie, file di configurazione, dataset e tutti i file che vengono utilizzati dal sistema nel momento dell'esecuzione. Sono la struttura più piccola che è possibile trovare all'interno di un sistema basato su ROS.

A livello di filesystem il package è rappresentato da una cartella. La struttura al suo interno comprende alcune sottocartelle per gestire gli elementi fondamentali per il suo sviluppo, in particolare:

- include/package_name: contiene prevalentemente gli headers C++;
- msg/: cartella che contiene i file relativi alle tipologie di messaggi (*Message types*);
- src/package_name/: cartella che contiene i file sorgente;
- srv/: cartella che contiene le tipologie di servizi (vedi Service types più avanti)
- scripts/: cartella che contiene script eseguibili dal software;
- CMakeLists.txt: file estremamente importante per la compilazione del package tramite CMake;
- package.xml: file che contiene in formato XML la struttura del package
- CHANGELOG.rst: all'interno del file vengono inseriti changelogs relativi agli aggiornamenti, questi verranno utilizzati dalle API di ROS all'interno dei file binari e durante la creazione della pagina Wiki del pacchetto;

È possibile generare automaticamente un pacchetto con una struttura iniziale già precompilata tramite l'utility di catkin *catkin_create_pkg*, in particolare:

```
#!/bin/bash
catkin_create_pkg <package_name> [depend1] [depend2]
```

Metapackages

I *metapackage* sono strutture specializzate che hanno come unico compito quello di rappresentare un gruppo di package che hanno caratteristiche comuni fra loro [12]. Nei progetti iniziati in versioni meno recenti di ROS e successivamente aggiornati, i metapackage possono essere anche il risultato di una conversione dei vecchi *stacks* che svolgevano una funzione simile.

Package manifests

I *package manifest* sono rappresentati da file XML che contengono in forma di metadati le caratteristiche e le informazioni riguardanti il package che li contiene. Sono inclusi in questi file il nome del package, la sua versione, le sue dipendenze, informazioni sulla licenza con il quale è distribuito il software e molto altro.

Esiste una serie di tag che devono essere obbligatoriamente inclusi nel manifest, e sono:

- <name> : il nome del pacchetto;
- <version> : la versione del pacchetto;
- <description> : la descrizione del pacchetto;
- <maintainer> : i nomi delle persone che supportano il pacchetto;
- <license> : la licenza sotto la quale è rilasciato il software (GPL, BSD, ASL, ecc);

Message types

I *message type* definiscono la struttura dei messaggi inviati da ROS [13]. Ogni file, con estensione *.msg*, rappresenta un tipo diverso di messaggio. All'interno dei file ogni riga rappresenta un campo del messaggio. Ogni riga a sua volta contiene due colonne: la prima relativa al tipo di dato del campo (Int32/int (C++/Phyton), bool, string, time, ecc), la seconda il nome.

È possibile assegnare all'interno di questi file dei valori ai campi, in questo caso si parla di costanti. Esempio di msg file (C++) :

```
1 Int32 distance
2 Int8 angle
3 string CONST_STR = "test"
```

Service types

I *service type* sono file che definiscono la struttura delle richieste e delle risposte per i servizi (srv) di ROS [14]. Al momento dell'esecuzione del software i file *.srv* sono utilizzati dalla libreria *roscpp* per generare codice C++ che, in base al contenuto del file originale, conterranno le definizioni dei servizi, l'implementazione delle richieste di servizio e, in ultimo, le risposte alle richieste.

La struttura C++ generata sarà simile a questa:

```
1  #Richieste costanti
2  int8 FOO=1
3  int8 BAR=2
4  #Richieste dipendenti dai campi
5  int8 foobar
6  another_pkg/AnotherMessage msg
7  ---
8  #Risposte costanti
9  uint32 SECRET=123456
10 #Risposte dipendenti dai campi
11 another_pkg/YetAnotherMessage val
12 CustomMessageDefinedInThisPackage value
13 uint32 an_integer
```

3.1.2 Computation graph level

Questo livello comprende un sistema di rete peer-to-peer che viene utilizzato per la comunicazione e la computazione di dati fra i vari nodi che compongono il grafo.

Nodes

I *node* sono processi che svolgono le funzioni principali del sistema e si occupano di processare i dati. Riprendendo il concetto di modularità del sistema, ogni nodo sarà relativo ad una sola specifica funzionalità. ROS infatti scoraggia la creazione di nodi "onnipotenti" che svolgono tante funzioni proprio per rendere il sistema meglio manutenibile, riusabile e chiaro [15]. I nodi solitamente sono sviluppati attraverso l'utilizzo di librerie come roscpp, rospy e roslisp che permettono la programmazione, rispettivamente, in C++, Python e Lisp. Sono in fase sperimentale tantissime altre librerie che permettono al programmatore di utilizzare Java (rosjava), NodeJS (rosnodejs), Go (rosgo), Lua (roslua) e tanti altri linguaggi di programmazione.

Master

Il *master* in ROS si occupa prima di registrare nuovi nodi all'interno della rete, poi di gestire la connessione fra i nodi del grafo, provvedendo ad instradare messaggi e permettendo l'accesso da parte di un nodo ai servizi di un altro [16]. È il cuore del software e può essere attivo solo un master per volta. È possibile avviarlo attraverso il comando *roscore* oppure lanciarlo in modo automatico all'avvio di un nodo attraverso la corretta implementazione del file.

Parameter server

Il *parameter server* è fondamentalmente un componente del master, permette di condividere in modo pubblico con tutti i nodi determinate configurazioni accessibili via network API. Anche se un sistema non estremamente performante è comunque utile per la fase di test del software [17].

Messages

I nodi del grafo comunicano attraverso lo scambio di messaggi. Questi possono essere semplici e di tipo primitivo (integer, float, string, char, ecc) oppure array o addirittura più complessi, con strutture simili a quelle viste in C [18].

Topics

I *topic* sono bus identificati tramite un nome proprio ed univoco che permettono lo scambio di messaggi fra i nodi [19]. Essi implementano un meccanismo di pubblicazione e sottoscrizione: i nodi possono essere publisher e/o subscriber nel caso siano predisposti per inviare o ricevere messaggi. La divisione fra chi produce dati e chi li utilizza è netta e separata tramite policy di anonimato fra i nodi. Ogni topic può avere un numero di messaggi massimo da tenere in coda nel caso si accumulassero, quelli in eccesso non vengono aggiunti alla coda e persi.

Esempio di pubblicazione di un messaggio da parte di un nodo all'interno di un topic:

```
1  //Advertise del topic sul quale si andrà a pubblicare e setting
2  //del numero massimo di messaggi da tenere in coda (in questo caso 5)
3  ros::Publisher pub = nh.advertise<std_msgs::String>("topic_name", 5);
4
5  //Creazione del messaggio
6  std_msgs::String str;
7  str.data = "Hello world!";
8
9  //Pubblicazione del messaggio
10 pub.publish(str);
```

Esempio di sottoscrizione da un topic da parte di un nodo:

```
1  //Definizione metodo callback all'arrivo del messaggio
2  void callback(const std_msgs::StringConstPtr& str)
3  {
4  ...
5  }
```

```
6
7   ...
8   /*
9   * Sottoscrizione del nodo al topic e setting del metodo di callback:
10  * ogni volta che un nuovo messaggio viene pubblicato sul topic, questo
11  * sarà ricevuto e inviato come parametro al metodo callback
12  */
13  ros::Subscriber sub = nh.subscribe("my_topic", 1, callback);
```

Services

I *service* sono uno strumento di comunicazione tra nodi di tipo bidirezionale. Si tratta di un meccanismo che estende quello dei *message* con la possibilità non solo di inviare dei comandi ad uno specifico nodo ma anche di restare in ascolto e ricevere una risposta strutturata da esso [20]. Ogni servizio viene prima descritto all'interno di un file *.srv* dove vengono indicati oltre al nome del servizio anche i parametri ed il tipo di dato di ritorno (vedi Service type).

All'interno del nodo *server* il servizio è rappresentato da una funzione che prende come input due puntatori ad oggetti della classe del server: uno includerà al suo interno i parametri della funzione (Request), l'altro invece raccoglierà il valore di ritorno (Response).

Un esempio di servizio messo a disposizione da un nodo server:

```
1   //Funzione associata al servizio, in questo esempio banale somma 2 interi
2   bool add(beginner_tutorials::AddTwoInts::Request  &req,
3            beginner_tutorials::AddTwoInts::Response &res)
4   {
5     res.sum = req.a + req.b;
6     ROS_INFO("request: x=%ld, y=%ld", (long int)req.a, (long int)req.b);
7     ROS_INFO("sending back response: [%ld]", (long int)res.sum);
8     return true;
9   }
10
11  int main(int argc, char **argv)
12  {
13    ...
14
15    //Creazione e \textit{advertise} del servizio in modo che sia visibile agli altri nodi
```

```
16   ros::ServiceServer service = n.advertiseService("add_two_ints", add);
17
18   ...
19
20   return 0;
21 }
```

Il client per richiamare il servizio non dovrà far altro che dichiarare ed impostare correttamente l'oggetto ClientService assegnandogli i giusti parametri e richiamare il servizio utilizzando la funzione *call* messa a disposizione dalla classe dell'oggetto.

Un esempio di chiamata da parte di un client:

```
1  ros::NodeHandle n;
2
3  //Dichiarazione del client
4  ros::ServiceClient client =
5  n.serviceClient<beginner_tutorials::AddTwoInts>("add_two_ints");
6
7  //Dichiarazione oggetto servizio
8  beginner_tutorials::AddTwoInts srv;
9
10 //Assegnamento parametri da inviare al servizio
11 srv.request.a = atoll(argv[1]);
12 srv.request.b = atoll(argv[2]);
13
14 //Chiamata al servizio e if condizionato sul valore di ritorno
15 if (client.call(srv))
16 {
17     //In caso di successo, stampa del campo sum dell'oggetto response
18     ROS_INFO("Sum: %ld", (long int)srv.response.sum);
19 }
20 else
21 {
22     ROS_ERROR("Failed to call service add_two_ints");
23     return 1;
24 }
```

Bags

Le *bag* rappresentano il sistema con il quale ROS salva log e tiene traccia di tutti i messaggi scambiati all'interno di un topic [21]. Il tool *rosbag*, una volta associato ad un topic, salva ogni messaggio scambiato all'interno di un relativo file in estensione *.bag*. È molto utile per memorizzare i dati provenienti dai sensori poiché permette allo sviluppatore la creazione di una sorta di "scatola nera" del robot. ROS mette a disposizione anche un tool di playback che permette di visualizzare e riprodurre i dati raccolti tramite un'interfaccia grafica (vedere rqt_bags). Esistono inoltre tool di terze parti che migliorano questa funzione integrandola con ulteriori servizi, ad esempio quelli di Amazon AWS.

3.1.3 Community level

ROS mette a disposizione degli sviluppatori le adeguate risorse per un costante scambio di informazioni ed idee. Attraverso un blog sempre attivo ed una wiki piuttosto vasta, ognuno può porre quesiti, rispondere ad altri utenti cercando di aiutarli oppure rendere pubblici i propri sorgenti. Questo è estremamente importante per un progetto come ROS, che deve una parte della sua fama proprio a questa sua filosofia dove la community è posta al centro. ROS fornisce una Wiki, un Blog e un portale Q&A a disposizione della community per interagire, scambiare risorse ed imparare il funzionamento di ROS.

ROS Wiki

La wiki viene mantenuta sia dallo staff di ROS che dagli utenti registrati che sono in grado di creare e modificare pagine relative a qualsiasi parte del sistema [22]. La lettura della wiki è, come per ogni grande progetto, l'approdo iniziale di qualunque sviluppatore e proprio per questo risulta molto semplice da consultare e ricca di contenuti ad ogni livello di esperienza. All'interno della Wiki sono inseriti anche i package creati dalla community.

Blog

Il Blog è il luogo dove le novità sul mondo di ROS vengono pubblicate regolarmente dallo staff per tenere aggiornata la community su prossimi update, conferenze e finanziamenti da parte degli investitori . Il sito serve soprattutto a mantenere un contatto fra il team di ROS e la sua community, in modo che quest'ultima conosca ad ogni passo la rotta del progetto.

ROS Q&A

Forse il portale più rilevante, soprattutto per i nuovi utenti, è quello Q&A. Il sito è molto semplice e risulta molto simile a Stack Overflow: si possono porre quesiti riguardanti progetti personali oppure sul funzionamento di alcune parti del sistema ROS, sulle API e sui package che sono all'interno della wiki e via dicendo. A rispondere a queste domande sono direttamente altri utenti registrati che hanno maggiore conoscenza oppure semplicemente hanno già incontrato e risolto lo stesso problema in altre situazioni. La forza di questo portale è ancora una volta una community molto attiva che non tarda mai a dare risposte e consigli. Questo ha permesso al portale di diventare come una seconda Wiki, o se vogliamo, una Wiki con risposte più immediate e precise sul problema.

3.2 ROS Tools

Come detto all'interno del primo capitolo, ROS oltre a fornire allo sviluppatore potenti API ed una solida architettura per lo sviluppo, si occupa di distribuire strumenti per analisi e debug del sistema. Questi risultano eccezionalmente utili a chi programma, ma in particolar modo a chi testa il software, sia in ambiente simulato che in ambiente reale.

3.2.1 RVIZ

RVIZ è sicuramente il più conosciuto e versatile dei tool, esso permette di visualizzare all'interno di uno spazio tridimensionale qualsiasi dato il software pubblichi attraverso i suoi topic [23]. I dati provenienti dai sensori, il modello stesso del robot sono alcune delle cose più essenziali che tramite un'interfaccia

grafica davvero intuitiva è possibile rendere visibile ed analizzabile. Oltre alla parte di osservazione ed analisi, RVIZ mette a disposizione anche dei semplici pulsanti con la quale è possibile inviare al sistema ROS in esecuzione dei messaggi relativi alla navigazione e alla telemetry del modello.

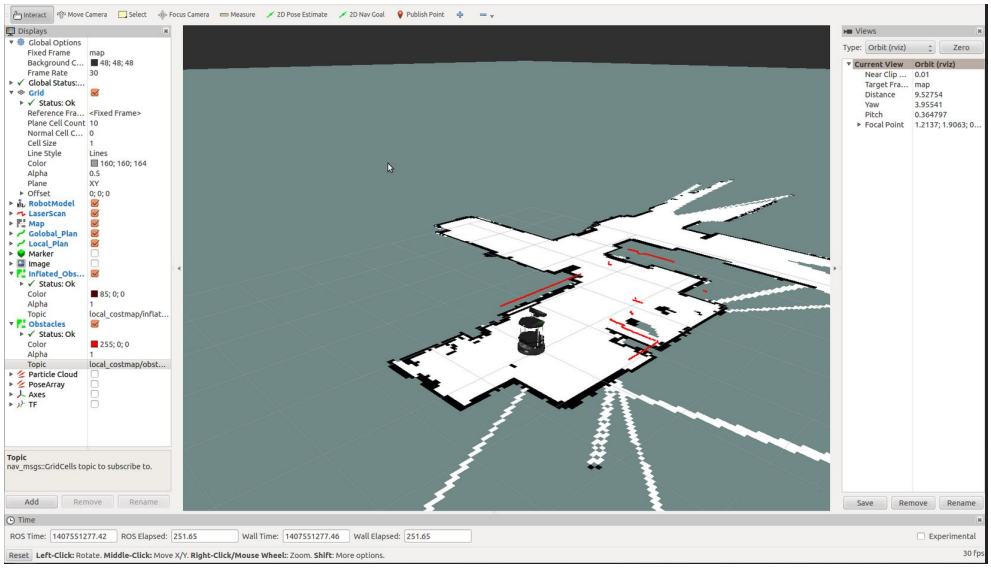


Figura 3.1: Esempio di visualizzazione dell'interfaccia grafica del tool RVIZ [24]

3.2.2 ROSBag e RQT_BAG

ROSbag è un tool a linea di comando che permette, attraverso delle api C++ o Python, di registrare e scrivere su file qualsiasi cosa stia succedendo all'interno del sistema ROS [25]. Input provenienti dai sensori, output di funzioni e dati di qualsiasi tipo possono essere memorizzati per poi essere analizzati e riprodotti. Proprio della riproduzione di questi dati si occupa RQT_BAG, un tool ad interfaccia grafica che permette di caricare, visualizzare e analizzare ogni "bag" creata, fornendo così un meccanismo simile alle scatole nere.

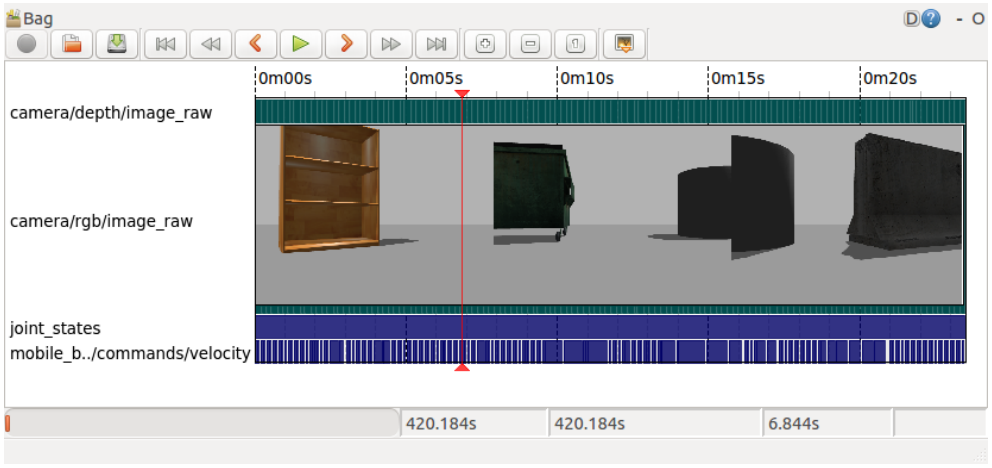


Figura 3.2: Esempio di visualizzazione dell'interfaccia grafica del tool RQT_BAG [26]

3.2.3 RQT_GRAPH

RQT_GRAPH è un tool grafico che permette di visualizzare, grazie ad un'interfaccia grafica, i processi attivi durante l'esecuzione [27]. Lo strumento risulta molto utile lasciando così rilevare all'utente anomalie nello scambio di dati attraverso i topic e fornendo una mappa dinamica del sistema in fase di esecuzione. Il tool inoltre mette a disposizione statistiche relative a particolari algoritmi, come il PID per esempio.

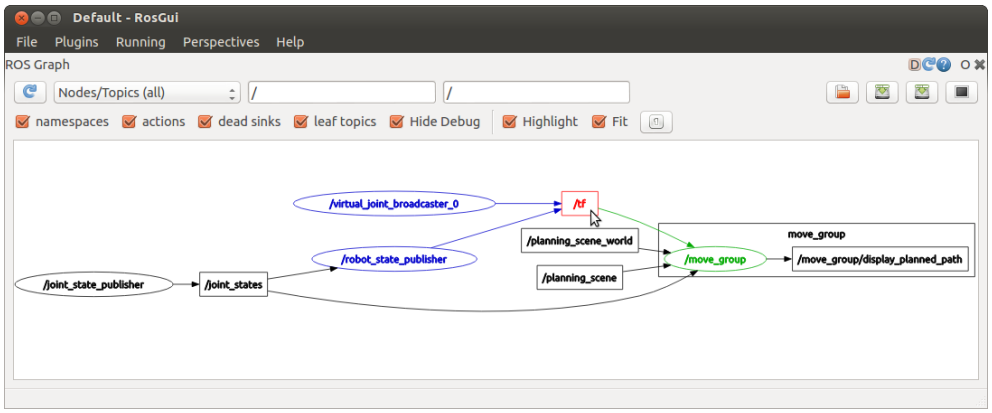


Figura 3.3: Esempio di visualizzazione dell'interfaccia grafica del tool RQT_GRAPH [28]

3.3 Programmazione di un robot in ROS: Un esempio

All'interno dei capitoli precedenti si è parlato di cosa sia la robotica di servizio, di come in questi ultimi anni sia cresciuta esponenzialmente e come ROS rappresenti una soluzione molto efficiente ed utilizzata per produrre applicazioni per questo nuovo mercato. Si è cercato inoltre di analizzare più nel dettaglio come fosse strutturata l'architettura di un applicativo ROS e come poter progettare e sviluppare un prodotto così complesso ed articolato come un robot utilizzando al meglio tutti gli strumenti messi a disposizione dello sviluppatore da parte del middleware.

Proprio alla luce di tutto ciò che fino ad ora è stato presentato, l'obiettivo di questo esempio è quello di concretizzare tutta questa teoria attraverso il progetto di un robot all'interno di un ambiente simulato.

Si intende così sviluppare un robot che sia in grado di muoversi in modo dinamico ed esplorare un ambiente casuale, evitare gli ostacoli sul proprio cammino senza collidere con essi ed infine costruire una mappa bidimensionale dell'ambiente rilevato.

Il robot, di piccole dimensioni, sarà dotato di due ruote e di uno chassis molto semplice. Grazie alle due ruote esso potrà muoversi anche sul posto, simulando quello che è il comportamento dei robot domestici per la pulizia della casa.

Lo spazio che circonda il soggetto comprenderà oggetti statici e dinamici, di dimensioni e forme differenti, per simulare al meglio un ambiente realistico.

La mappa generata dall'esplorazione sarà salvata e caricata rispettivamente ad ogni spegnimento ed accensione del robot. Le modalità con cui è possibile interagire con il robot saranno due:

- Esplorazione automatica
- Esplorazione manuale

Durante l'esplorazione automatica il robot avrà il compito di esplorare in modo totalmente autonomo un'area all'interno di un perimetro delineato in partenza e personalizzabile ad ogni utilizzo.

Nella seconda modalità, quella manuale, un operatore avrà la possibilità di indicare punto per punto dove il robot dovrà andare utilizzando un'interfaccia grafica di controllo.

Anche in questo caso sarà attivo il meccanismo che permette al robot di evitare ostacoli e creare la mappa dell'ambiente.

Il robot avrà un'autonomia legata ad una batteria simulata che verrà consumata ad ogni movimento e sarà ricaricata quando esso sarà sulla piattaforma di ricarica, la stessa che verrà presa come riferimento e punto di origine di coordinate (0,0).

3.4 Progettazione

3.4.1 Ambiente di simulazione

Per la creazione dell'ambiente di simulazione è stato scelto Gazebo. Semplice da utilizzare e largamente supportato, è un ottimo software che riesce a ricreare in modo accurato l'ambiente reale (forza di gravità, collisioni, ecc).

Il robot si muoverà all'interno di una piccola stanza creata per testare le sue capacità e mettere alla prova la sua abilità nel muoversi all'interno di spazi stretti e di eseguire manovre con margini minimi di errore.

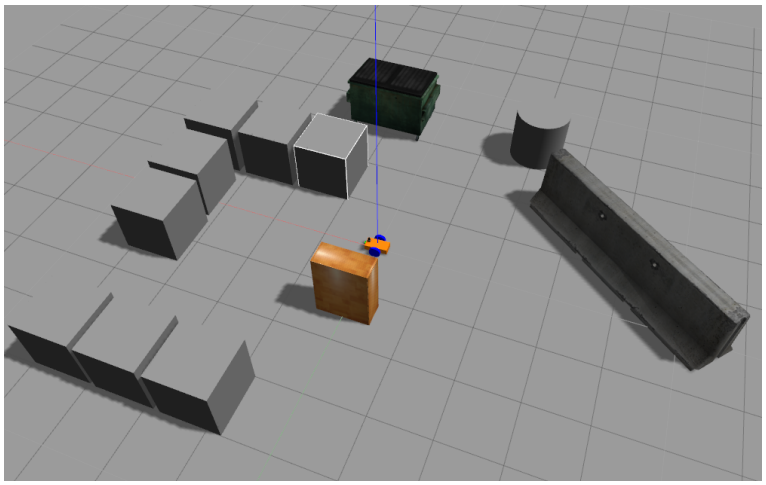


Figura 3.4: Visualizzazione dell'embiente simulato visto da Gazebo

3.4.2 Modello del robot

Il robot è vagamente ispirato ad un piccolo robot casalingo per l'aspirazione degli ambienti.

Il corpo è essenzialmente un parallelepipedo di piccole dimensioni. Sulle facce laterali del poligono sono montate due ruote che permettono al robot di muoversi in ogni direzione.

Sulla parte parallela al terreno sono montate due sfere che permettono al robot una stabilità costante ed al contempo una rotazione a 360 gradi. Infine, sulla sommità è posizionato il laser scanner. Posto al di sopra di ogni parte del modello, ha una visuale libera ed è in grado di sfruttare completamente la sua rotazione.

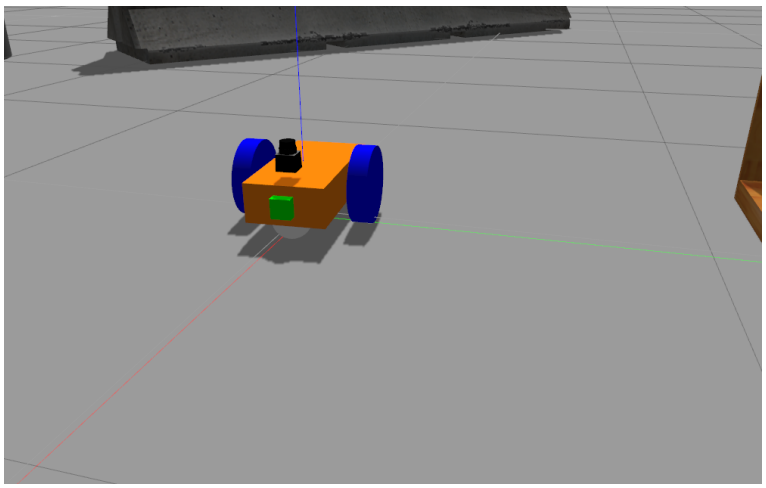


Figura 3.5: Modello del robot usato per la simulazione visto da Gazebo

3.4.3 Rilevazione dell'ambiente

L'ambiente viene scansionato attraverso un laser scanner bidimensionale che, come detto precedentemente, è stato posto sulla sommità del robot proprio per non avere ostacoli che ne limitassero il campo visivo. Questo aumenta il flusso di dati da gestire ma migliora la qualità del path planning, oltre che velocizzare il processo di composizione della mappa. Il laser scanner 2D risulta molto semplice da implementare e richiede scarse risorse hardware rispetto ad uno tridimensionale, per questo in questo caso di studio è stato preferito.

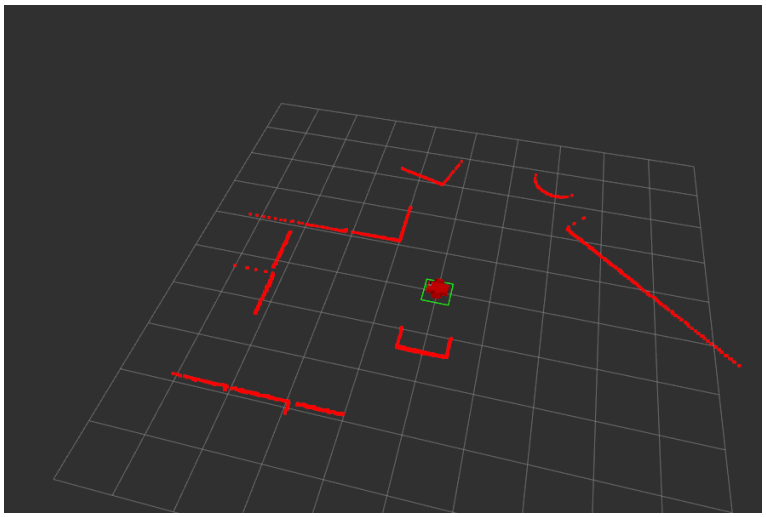


Figura 3.6: Visualizzazione tramite RVIZ dell'ambiente rilevato dal laser scanner montato sul robot

3.4.4 SLAM

Un passaggio fondamentale per poter muovere il robot in un ambiente è la costruzione di una mappa ed il posizionamento del robot all'interno di essa.

Gli algoritmi SLAM fanno proprio questo, ovvero, prendendo come input le rilevazioni dell'ambiente esterno e i dati riferiti all'odometria del robot in modo da costruire una mappa a partire da punti di riferimento che vengono aggiornati ogni volta che un dato in input viene ricevuto.

Tutti questi punti vengono confrontati in modo consecutivo e consentono al robot di orientarsi e costruire la mappa.

La scelta dell'algoritmo SLAM da utilizzare all'interno del progetto è stata fatta attraverso lo studio del documento [29] che mette a confronto alcuni dei piu noti algoritmi sviluppati.

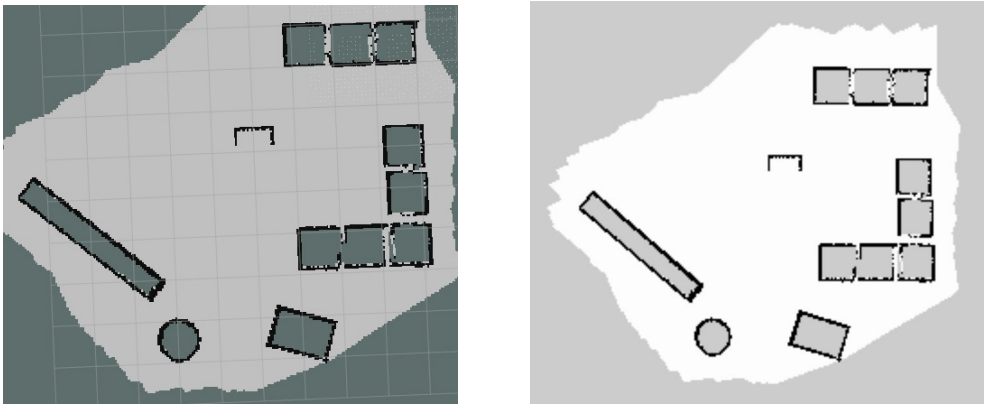
	HectorSLAM	Gmapping	KartoSLAM	CoreSLAM	LagoSLAM
Test 1	1.1972	2.1716	1.0318	14.75333	3.0264
Test 2	0.5094	0.6945	0.3742	7.9463	0.8181
Test 3	1.0656	1.6354	0.9080	7.5824	2.5236
CPU	6.1107%	7.0873%	5.4077%	5.5213%	21.0839%

Il più efficiente sia in termini di performance che di errori commessi durante i 3 test eseguiti da chi ha redatto il documento è stato Karto SLAM.

Pur non essendo il migliore, all'interno di questo caso di studio è stato adottato Gmapping. Questa scelta è stata dettata dalle performance strettamente comparabili con gli altri due algoritmi, Karto e Hector, ma con il vantaggio di avere un'implementazione più semplice, una documentazione ed un supporto da parte della community più ampi.

La localizzazione del robot all'interno della mappa avviene utilizzando i topic /map, /odom e /chassis. Il frame /map è il sistema con il quale la mappa viene orientata, è puntato alle coordinate di origine e viene preso come riferimento per calcolare la posizione del robot e costruire la mappa. Il frame /odom invece è ciò che contiene i dati sull'odometria del robot (orientamento, inclinazione, velocità, ecc). Quando l'algoritmo SLAM è attivo la posizione e i dati del frame /odom vengono aggiornati e corretti attraverso la trasformazione dei dati provenienti dal frame /map.

Il lavoro dell'algoritmo si traduce nella produzione di una mappa composta da una griglia, detta occupancy grid, dove le celle vengono identificate come occupate, se in corrispondenza di un ostacolo, libere, se esplorate e in assenza di ostacoli, o "non conosciuto" per tutte le celle che non sono state ancora esplorate. Nella prima immagine sotto (a) è possibile vedere i tre valori rappresentati rispettivamente con il colore nero, grigio o verdastro, mentre nella seconda (b) è possibile visualizzare la stessa mappa esportata a fine dell'esplorazione con i contorni degli oggetti colorati di nero, le aree esplorate di bianco e quelle sconosciute di grigio.



(a) Visualizzazione Gmapping (b) Visualizzazione mappa esportata

Figura 3.7: Visualizzazioni a confronto

3.4.5 Navigazione

La navigazione è l'attività principale che svolge il robot insieme alla creazione di una mappa.

Per quest'attività è stato implementato quello che viene definito ROS Navigation Stack: un sistema di navigazione basato sull'algoritmo dei cammini minimi ideato da Dijkstra.

L'algoritmo comparato con altri, come per esempio Navigation Dodicch, risulta migliore dal punto di vista dell'agilità con cui il robot cambia direzione e reagisce ad eventuali cambiamenti dell'ambiente esterno. Il navigation stack è formato da :

- un pianificatore globale che ha lo scopo di calcolare un percorso per collegare un punto iniziale ad un punto finale all'interno della mappa, ed è la componente basata su Dijkstra.

- un pianificatore locale che ha lo scopo di calcolare le velocità da inviare al robot per seguire il percorso calcolato dal pianificatore globale e allo stesso tempo evitare gli ostacoli mobili.

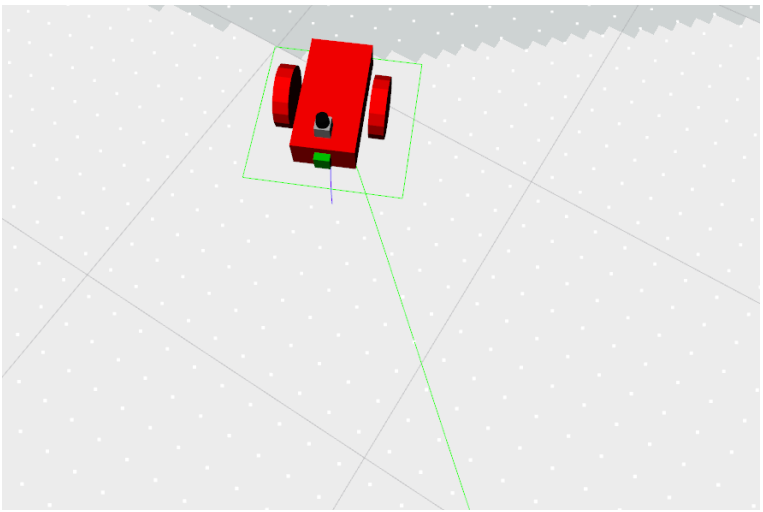


Figura 3.8: Visualizzazione tramite RVIZ dell'azione a runtime del pianificatore globale (linea verde) e di quello locale (linea blu)

Per il corretto funzionamento dell'algoritmo, Navigation Stack richiede una configurazione.

Questa attività avviene per entrambe le mappe relative ai pianificatori, locale e globale, ed ha il compito, attraverso la modifica di decine di parametri, di perfezionare ed affinare il comportamento del robot durante la navigazione in base alla forma del soggetto, alla tipologia della mappa e così via.

Oltre al settaggio delle mappe è necessaria la configurazione del plugin *Inflaction Costmap* e di conseguenza del comportamento dei pianificatori, in particolare di quello locale. Questo permette l'assegnazione di costi variabili a ciascuna cella della griglia creata dall'algoritmo di *SLAM* e di conseguenza la creazione di percorsi più efficienti da parte dei pianificatori.

Per la creazione delle mappe è necessaria una configurazione attraverso il settaggio di parametri specifici.

Per fare ciò è stato necessario suddividere in 3 file i parametri: un primo file per i parametri comuni fra le mappe, il secondo per quelli esclusivamente relativi alla mappa globale ed infine il terzo per quelli relativi alla mappa locale.

I parametri della configurazione comune delle due mappe sono:

- global_frame: /map, identifica il nome del sistema di riferimento della mappa;

- robot_base_frame: /chassis, identifica il nome del sistema di riferimento del robot;
- obstacle_range: 2.5, identifica la distanza espressa in metri entro quale i valori del laser scanner vengono presi in considerazione;
- raytrace_range: 3.0, identifica la distanza in metri entro quale aggiornare lo spazio libero;
- footprint: [[0.25, 0.25], [0.25, -0.25], [-0.25, -0.25], [-0.25, 0.25]], identifica l'area che il robot occupa all'interno della mappa;
- observation_sources: laser_scan_sensor, identifica la sorgente dalla quale vengono acquisiti i dati sull'ambiente esterno, ovvero se di tipo laser o RGB-D;
- laser_scan_sensor: sensor_frame: laser_frame, data_type: LaserScan, topic: /scan, marking: true, clearing: true, associa i dati del sensore laser scanner alla configurazione;
- inflation_radius: 0.6, letteralmente "raggio di inflazione", è il raggio espresso in metri che circonda un oggetto e che viene suddiviso in aree di costo per ottimizzare la navigazione nei pressi degli ostacoli;

I parametri specifici la costruzione della mappa globale (m.g.) e della mappa locale (m.l.) sono i seguenti:

- update_frequency: 5.0 (m.g.), 5.0 (m.l.), la frequenza espressa in Hz di aggiornamento della mappa;
- publish_frequency: 5.0 (m.g.), 5.0 (m.l.), la frequenza espressa in Hz di pubblicazione della mappa;
- transform_tolerance: 0.5 (m.g.), 0.25 (m.l.),
- width: 30.0 (m.g.), 20.0 (m.l.), la larghezza espressa in metri della mappa di costo;
- height: 30.0 (m.g.), 20.0 (m.l.), l'altezza espressa in metri della mappa di costo;

- origin_x: -15.0 (m.g.), -10.0 (m.l.), la coordinata X per il punto di origine della mappa di costo;
- origin_y: -15.0 (m.g.), -10.0 (m.l.), la coordinata Y per il punto di origine della mappa di costo;
- static_map: true (m.g.), false (m.l.),
- rolling_window: (m.g.), true (m.l.),
- resolution: 0.1 (m.g.) 0.05 (m.l.), la dimensione espressa in metri di ogni lato delle celle di costo presenti sulla mappa;
- Infine la configurazione dei pianificatori, in particolare di quello locale. I parametri settati sono i seguenti:
- max_vel_x: 0.2, la velocità lineare massima espressa in m/s che il robot è in grado di raggiungere
- min_vel_x: 0.1, la velocità lineare minima espressa in m/s che il robot è in grado di raggiungere
- max_vel_theta: 0.35, la velocità di rotazione massima espressa in rad/s;
- min_vel_theta: -0.35, la velocità di rotazione minima espressa in rad/s;
- max_rotational_level: 0.25, la velocità di rotazione massima sul posto espressa in rad/s
- min_in_place_vel_theta: 0.25, la velocità di rotazione minima sul posto espressa in rad/s
- acc_lim_theta: 0.25
- acc_lim_x: 2.5
- acc_lim_Y: 2.5
- occdist_scale: 0.20, valore utilizzato durante il calcolo delle traiettorie, rende meno vantaggioso scegliere percorsi che passano a fianco di oggetti

- holonomic_robot: true, imposta il comportamento del robot in modo che sia più restrittivo nel seguire la traiettoria calcolata

- xy_goal_tolerance: 0.15, la tolleranza in metri sulla posizione di arrivo

- yaw_goal_tolerance: 0.25, la tolleranza in radianti dell'angolo della posizione di arrivo

- cost_factor: 0.55, fattore di moltiplicazione per ogni costo assegnato

- neutral_cost: 66

- lethal_cost: 253

Alcuni di questi parametri sono stati più cruciali di altri durante lo sviluppo del robot, ad esempio :

- inflaction_radius, parametro molto importante per permettere al robot di passare attraverso punti della mappa con poco spazio di manovra. All'inizio dello sviluppo il valore impostato troppo alto faceva sì che gli ostacoli occupassero molta più area del dovuto. Una volta abbassato ha permesso al robot di muoversi meglio anche in situazioni difficili, mantenendo però una certa area di sicurezza che impedisce la collisione.

- xy_goal_tolerance e yaw_goal_tolerance, introdotti dopo aver riscontrato che il robot arrivato a destinazione iniziava una rotazione infinita su se stesso. Questo perchè molto spesso non è possibile centrare in modo assolutamente perfetto l'obiettivo e questi due parametri garantiscono quel poco di tolleranza essenziale per non incorrere in questo tipo di problemi.

- holonomic_robot, sebbene il robot sin dall'inizio funzionasse, non seguiva correttamente i percorsi pianificati. Per ovviare a questo problema gli è stato assegnato un comportamento più restrittivo in modo da essere più preciso nei movimenti.

3.4.6 Inflation

L'Inflation è il processo con il quale l'area di un oggetto viene propagata nelle sue immediate vicinanze ed a ognuna delle celle occupate da quest'area viene assegnato un costo in valore numerico che diminuisce con l'aumentare della distanza da esso.

A questo scopo si possono definire 5 livelli di costo che interagiscono con il calcolo del percorso :

- "Lethal" è il costo assegnato al punto su cui l'ostacolo è concretamente posizionato. Il robot in condizioni normali non utilizza mai questi punti per il suo percorso, poiché avverrebbe una collisione più che certa. (punti di colore viola-fucsia nell'immagine)

- "Inscribed" è il livello di costo assegnato nelle primissime vicinanze dell'ostacolo. La distanza che intercorre fra l'inizio di questa zona ed il suo limite esterno è proporzionale all'ampiezza del robot. All'interno di quest'area il robot colliderebbe con l'ostacolo nel momento in cui il suo centro si trovasse in quest'area. (punti di colore azzurrino nell'immagine)

- "Possibly circumscribed", simile al precedente "inscribed" ma con la differenza che la sua ampiezza è proporzionale ad un altro valore del robot che è il "circumscribed radius", ovvero l'area esterna al modello fisico del robot ma che comunque viene considerata come "zona cuscinetto" durante le manovre. (punti di colore viola-scuro nell'immagine)

- In questa zona la probabilità di collidere con l'oggetto esiste ma non è sicura, quindi, i punti all'interno di quest'area presentano un costo non conveniente in situazioni normali. Questo livello viene utilizzato anche per escludere dal passaggio alcune aree, senza che ci sia un vero e proprio ostacolo, solamente come zona non accessibile

- "Freespace" con un costo pari a zero è il livello che raccoglie tutti i punti in un'area dove il rischio di collidere con l'oggetto non esiste e dove il passaggio è sicuro. (punti di colore bianco nell'immagine)

- "Unknown" a questo livello appartengono le zone inesplorate. Esistono altri livelli di costo intermedi fra "Possibly circumscribed" il cui costo

è intermedio e proporzionale alla distanza dall'ostacolo. (es. punti di blu viola nell'immagine). In generale possiamo dire che più il punto sarà vicino all'ostacolo, più il costo per raggiungerlo/transitarci sopra sarà elevato, in alcuni casi, tanto da non risultare accessibile.

Questo sistema di costi viene utilizzato nel momento in cui il robot disegna il tracciato per arrivare ad un determinato punto della mappa, in questo caso, il robot utilizzerà quello che, sommando il costo di ogni singolo punto, avrà il costo totale più basso, in modo da prendere il percorso che sia più breve ma con un rischio di collisione minore possibile. A livello grafico è inoltre possibile notare questo sistema tramite il diverso colore delle aree intorno ad un ostacolo.

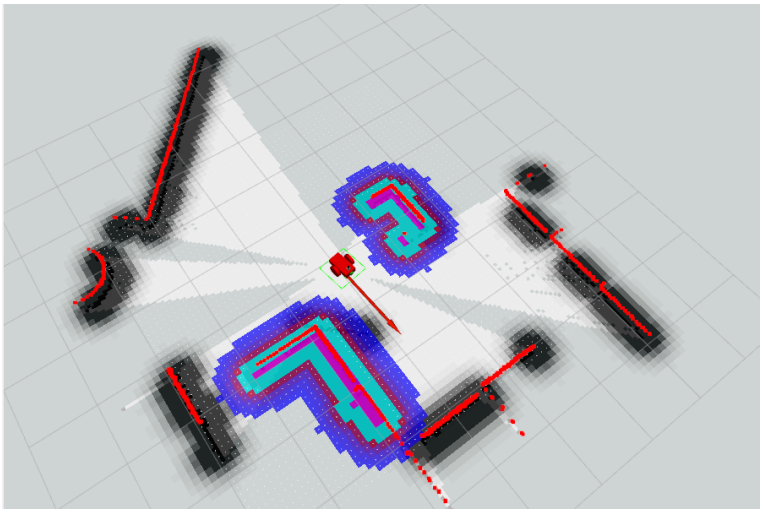


Figura 3.9: Visualizzazione tramite RVIZ della colorazione delle aree adiacenti agli ostacoli rilevati dal software

3.4.7 Esplorazione autonoma

L'esplorazione autonoma è il processo con il quale il robot è in grado di muoversi senza un controllo diretto da parte dell'utente e costruire in modo autonomo una mappa dell'ambiente esplorato. Per far si che questo accada è necessario utilizzare un algoritmo per l'esplorazione. In forma di nodo è disponibile Frontier Exploration, un pacchetto sfrutta il navigation stack di ROS configurato precedentemente e permette quindi una facile implementazione.

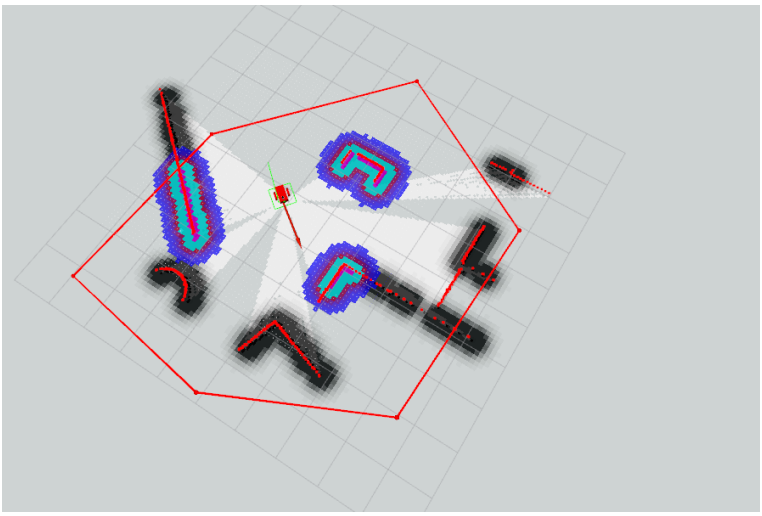


Figura 3.10: Visualizzazione tramite RVIZ dei bordi delle frontiere (frontiers) appena impostate.

Frontier Exploration è stato preferito ad altri proprio per questo aspetto, ovvero, la condivisione con il navigation stack della configurazione e l'utilizzo di quest'ultimo come algoritmo di navigazione, caratteristica che a mio parere si sposa benissimo con l'idea di riuso del codice. Presi in input dei punti chiamati *frontier* esso disegna un poligono all'interno del quale il robot esplorerà l'area.

3.4.8 Gestione della batteria

La gestione della batteria permette di simulare i consumi energetici del robot durante i suoi movimenti. Ogni volta che il soggetto è in movimento il livello della batteria viene decrementato di una quantità direttamente proporzionale alla velocità a cui sta andando. Allo stesso tempo, ogni volta che il robot cambia posizione viene stimato il consumo per il suo ritorno alla stazione di ricarica, situata al centro della sua mappa. Quando la stima coincide con il livello di batteria corrente, più una percentuale di scarto, il soggetto inverte la rotta e torna verso la stazione dove sarà caricato nuovamente.

Capitolo 4

Caso di studio reale

Insieme ad un gruppo di altre due persone, conosciute in ambito lavorativo, è nata l'idea di costruire, a partire dall'idea di uno dei due, un prodotto che migliorasse la vita della manovalanza in ambito agricolo ed in generale ovunque si spendessero inutilmente energie per spostare pesi attraverso la manovalanza.

4.1 Obiettivi

L'obiettivo del progetto era quello di sviluppare un robot cingolato capace di inseguire un soggetto umano, detto operatore, per aiutarlo durante i lavori di spostamento delle merci, raccolta di frutta e verdura ed altri compiti simili che per un uomo risultano pesanti e faticosi ma che a volte non vengono svolti con altre macchine perché ingombranti o scomode. Il robot avrebbe dovuto avere un piano di carico autolivellante e mantenere una marcia coerente con quella dell'operatore. Sarebbe dovuto essere presente inoltre un GPS ed un modulo per la connessione internet tramite sim telefonica per il monitoraggio remoto della macchina attraverso un'interfaccia di controllo amministratore.

L'idea, partita dall'ambito agricolo, doveva essere quella di sviluppare una sorta di aiutante agile, di dimensioni ridotte, facile da utilizzare, molto più economico di quelli già presenti sul mercato e comodo per lavori dove l'impiego di macchine più grandi e costose sarebbe esagerato ed eccessivamente dispendioso.

4.2 Primi passi

Durante lo sviluppo della versione "Zero" non era ancora presente la consapevolezza di quanto fosse complesso l'intero progetto e per iniziare a prendere conoscenza, testare i primi movimenti e vedere come si comportasse la meccanica abbiamo scelto di utilizzare per la prototipazione la piattaforma Arduino. Questa garantiva facilità e rapidità di sviluppo per le parti iniziali (muovere il motore, prendere dati dai primi sensori) e costi di produzione estremamente bassi. L'architettura era davvero molto semplice e comprendeva solamente un Arduino, 2 driver controller per pilotare i due motori, uno per ogni cingolo, ed un modulo bluetooth con cui un'applicazione per Android inviava al robot dei comandi di base e dal quale poi riceveva una quantità minima di informazioni per eseguire analisi in tempo reale sullo stato del sistema. Per iniziare, i movimenti del robot erano piuttosto semplici e prevedevano una sola rampa di accelerazione lineare, sia per la marcia che per la svolta, come possibile notare qui sotto in questo snippet di codice dove è codificata la svolta verso destra del robot in questa fase dello sviluppo.

```
1  /* Se si e' impostata la rotazione del robot verso destra */
2  if(rightward_flag){
3
4      /*
5      * CONTROLLI SUL MOTORE DESTRO
6      * Controllo se il motore è già in movimento e la sua direzione
7      * è quella di marcia avanti
8      */
9      if(motor_right->get_pwm_value() > 0 &&
10         motor_right->get_current_direction() == MOTOR_DIRECTION_FORWARD){
11
12       /*
13       * Decremento la velocità di rotazione del motore destro per evitare
14       * bruschi scatti
15       */
16       this->pwm_to_set_right = motor_right->get_pwm_value() -
17       CONST_RAMP_ACCELERATION;
18
19       /*
20       * Controllo se il motore destro e' fermo e se la direzione del motore
```

```
21          * è impostata su marcia avanti
22          */
23        }else if(motor_right->get_pwm_value() <= 0 &&
24                motor_right->get_current_direction() == MOTOR_DIRECTION_FORWARD){
25
26            /* Inverto la marchia del motore poichè fermo */
27            motor_right->set_backward();
28
29          /*  Controllo se il motore è fermo e con la marcia impostata su indietro */
30        }else if(motor_right->get_pwm_value() >= 0 &&
31                motor_right->get_current_direction() == MOTOR_DIRECTION_BACKWARD){
32
33          /*
34          * Il motore è pronto per essere mosso indietro ed iniziare la manovra
35          * di rotazione del robot verso destra
36          */
37          this->pwm_to_set_right = motor_right->get_pwm_value() + CONST_RAMP_ACCELERATION;
38        }
39
40        /*
41        * CONTROLLI SUL MOTORE SINISTRO
42        * Controllo se il motore è già in movimento e la sua direzione
43        * è quella di marcia indietro
44        */
45        if(motor_left->get_pwm_value() > 0 &&
46            motor_left->get_current_direction() == MOTOR_DIRECTION_BACKWARD){
47
48          /*
49          * Decremento la velocità di rotazione del motore sinistro
50          * in modo daevitare bruschi scatti
51          */
52          this->pwm_to_set_left = motor_left->get_pwm_value() - CONST_RAMP_ACCELERATION;
53
54        }else if(motor_left->get_pwm_value() <= 0 &&
55                motor_left->get_current_direction() == MOTOR_DIRECTION_BACKWARD){
56
57            //Inverto la marchia del motore poichè fermo
58            motor_left->set_forward();
59
60        }else if(motor_left->get_pwm_value() >= 0 &&
61                motor_left->get_current_direction() == MOTOR_DIRECTION_FORWARD){
```

```
62
63              /*
64          * Il motore e' pronto per essere mosso in avanti ed iniziare
65          * la manovra di rotazione del robot verso destra
66          */
67          this->pwm_to_set_left = motor_left->get_pwm_value() + CONST_RAMP_ACCELERATION;
68      }
69  }
```

Lo sviluppo di questa parte di progetto fu relativamente semplice e permise a ciascuno di prendere confidenza con il progetto e pensare ai prossimi passi.

4.3 Rilevamento dell'ambiente

All'interno della versione versione "Uno" fu introdotto per la prima volta il concetto di rilevamento degli ostacoli e dell'operatore. Furono inseriti sul robot alcuni sensori per il rilevamento della distanza con tecnologia ad ultrasuoni, molto simili a quelli utilizzati sulle auto come assistenti alla manovra di parcheggio. Furono posizionati sulla parte anteriore e su ciascuno dei lati, in modo da rilevare gli ostacoli sia durante la marcia che durante la rotazione.

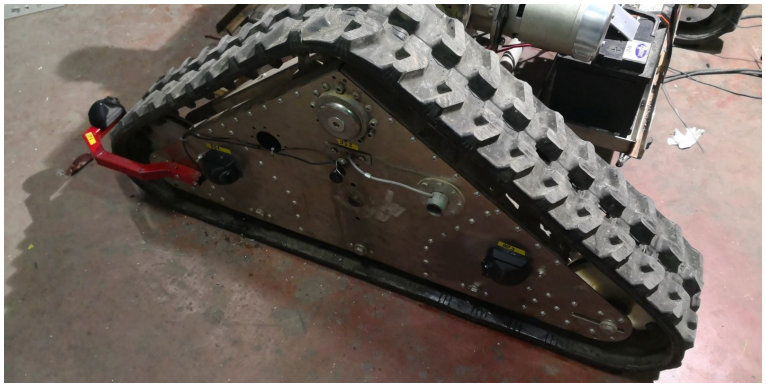


Figura 4.1: Foto della parte sinistra del robot dove è possibile notare i 3 sensori laterali e il sensore frontale sinistro

Alcuni sensori per la ricezione dei raggi infrarossi furono montati sulla parte anteriore per l'identificazione dell'operatore che a sua volta aveva dietro la schiena un emettitore. L'idea era quella di ridurre in angoli di circa 30

gradi il campo visivo dei ricevitori IR così che, una volta integrato il dato della rilevazione con le distanze registrate sulla parte anteriore, si potesse con un livello approssimativo di precisione capire dove si trovasse l'operatore. Il sistema era pensato per tenere l'operatore ad una distanza dal robot minima, sotto al metro, in modo che la precisione fosse maggiore possibile.

4.4 Problemi e prime soluzioni

Se i primi passi erano stati mossi con sufficiente semplicità, con l'aumentare del numero di componenti le prime difficoltà nella gestione del flusso d'informazione si fecero sentire. Una sola scheda Arduino che gestiva un numero sempre maggiore di sensori, iniziò presto a diventare un ostacolo per l'avanzamento del progetto. In particolare le difficoltà maggiori vennero riscontrate in relazione al numero di ricevitori infrarossi pilotati tramite la libreria IRremote. Su ogni scheda Arduino non è possibile, infatti, montare più di un singolo ricevitore infrarosso senza andare incontro a problemi legati ai timer della scheda. La libreria indicata, come tantissime altre, agisce proprio sul timer interno di Arduino per gestire il sensore e dopo numerose ricerche è stato necessario arrendersi all'evidenza che era obbligatorio aumentare il numero delle schede e la complessità dell'architettura. Così la nuova architettura prevedeva una scheda Arduino Nano per ogni ricevitore IR, 6 in totale, alla quale oltre allo stesso sensore era collegato un led che veniva illuminato in caso di eccitazione del sensore. Quando un sensore veniva "colpito" dal segnale IR emesso dall'emettitore posizionato sull'operatore, comunicava tramite seriale ad un Arduino Mega il suo ID. La scheda che lo riceveva a questo punto elaborava l'informazione e restituiva in output una direzione. Questa veniva inviata ad un secondo Arduino Mega il quale compito era quello di gestire la parte relativa ai motori, quindi, impostare la direzione corrispondente all'informazione ricevuta combinandola alle informazioni che quest'ultima riceveva dai sensori distanziometrici per il rilevamento degli ostacoli intorno al robot. La scelta di suddividere il carico di lavoro su 2 schede Arduino Mega venne presa a causa di numerose motivazioni, in particolare i driver controller, le seriali, i sensori di calcolo della distanza, i numerosi led e tanti altri sensori e piccoli attuatori agganciati al robot non era più possibile fisicamente attaccarli ad

una sola scheda per mancanza di pin, oltre a ciò la potenza computazionale di un solo Arduino non era più sufficiente a mantenere il sistema senza incorrere in rallentamenti e problemi.

4.5 Miglioramenti della comunicazione

A questo punto però sorse un secondo problema: l'affidabilità del collegamento seriale tra le schede Arduino. Le informazioni passate arrivavano spesso corrotte o disturbate, soprattutto quando i flussi di dati diventavano piuttosto intensi e la scheda spesso concatenava in modo errato i messaggi. Per ovviare a questi problemi fu introdotta una scheda Raspberry Pi. Questa veniva utilizzata come server centrale fra le principali piattaforme che fungevano da client, in questo modo le board non dovevano essere connesse direttamente fra loro come in un sistema peer-to-peer, evitando così tutti i problemi legati a questo tipo di architettura. Su Raspberry era installato Node-Red, software creato da IBM e scelto per l'estrema semplicità di utilizzo, la vasta gamma di plugin disponibili e perchè poco "esigente" in termini di risorse computazionali, fattore che si sposava benissimo con la nostra architettura. Il software è molto semplice e pensato per l'industria IOT, in particolare per automatizzare processi legati alla domotica. Il linguaggio di programmazione utilizzato è Javascript anche se molte azioni è possibile programmarle attraverso un editor visuale sfruttando plugin ufficiali o della community. Grazie proprio a questo passo in avanti si riuscì ad introdurre un database per memorizzare ed analizzare a posteriori i log provenienti dal robot ed un'interfaccia grafica stile dashboard per visualizzare i dati in tempo reale senza dover letteralmente impazzire nel leggere i dati scorrere dal monitor della seriale. Quello che emerse da questo upgrade fu che il punto debole della versione precedente era l'assenza di un sistema operativo che desse la possibilità di sviluppare ed installare software più complessi, così da analizzare i dati con più falicilità ed un'architettura molto debole, in particolare dal punto di vista della comunicazione e del trasferimento dati. Grazie alla possibilità di eseguire analisi migliorate sul flusso dei dati, fu possibile migliorare anche lo studio sui sensori e valutare l'acquisto di nuovi e di miglior fattura.

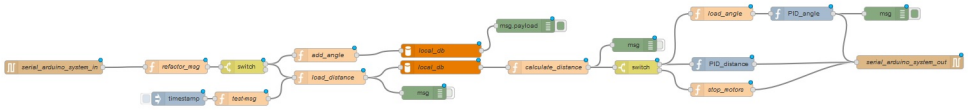


Figura 4.2: Visualizzazione del flow principale gestito da Node-Red

4.6 Affinamento dell'inseguimento

Aumentare il numero di sensori e la loro qualità di versione in versione significava una precisione maggiore ed un flusso sempre più elevato di dati da utilizzare all'interno del software e per eseguire analisi sul comportamento del robot. Questo almeno fino al raggiungimento dei limiti tecnici dei sensori stessi. I sensori ad ultrasuoni utilizzati per il rilevamento degli ostacoli, infatti, iniziarono con i test ad evidenziare problemi in relazione al rilevamento del vuoto ed al rimbalzo delle onde sonore su superfici fonoassorbenti, come per esempio gli indumenti. Accostando i sensori ad ultrasuoni con altri a rilevazione della distanza tramite laser infrarossi il problema veniva contenuto, ma solamente in ambienti chiusi. All'esterno degli edifici, infatti, il sole intenso e la rifrazione dell'aria in certi casi non facevano altro che aggravare la situazione e per un robot pensato per le condizioni, talvolta estreme, della campagna, non era di certo un problema secondario. Tutto ciò ovviamente ebbe ripercussioni negative sul sistema di inseguimento dell'operatore. L'algoritmo di modulazione della velocità, implementato attraverso lo sfruttamento del meccanismo PID, era fortemente basato sulla rilevazione della distanza e come è facile intuire, nel momento in cui i dati di input relativi alla distanza risultano non corretti e non corrispondenti alla realtà, il comportamento del robot a sua volta risulterà imprevedibile e non conforme a ciò che realmente è all'interno dell'ambiente.

Node-Red, pur rappresentando un passo in avanti importante rispetto al trasferimento crudo di stringhe fra le seriali delle schede, risultava però ancora troppo macchinoso e triviale da utilizzare. I dati arrivavano al server strutturati in JSON su Arduino tramite una libreria di terze parti, venivano convertiti in un oggetto Javascript e poi utilizzati. Ciò però risultava macchinoso, sia

perchè si trattava comunque di manipolazione di stringhe, sia perché i tipi di dati utilizzati non coincidevano fra i due sistemi. Quello di cui avremmo avuto bisogno a questo punto era uno scambio di dati tramite invio di oggetti in modo nativo, evitando il più possibile la conversione di stringhe utilizzando da entrambe le parti gli stessi tipi di dato. A parte i problemi individuati in questa versione relativi al comparto di scambio dati e ai sensori di rilevamento della distanza, venne introdotto un nuovo metodo per individuare con maggior precisione l'operatore. Il sistema era leggermente più complesso del precedente ma più raffinato: un ricevitore di raggi infrarossi fu montato su un servo motore che si muoveva a velocità costante sull'asse verticale. Il sensore era coperto da un cilindro chiuso dove solo uno spiraglio permetteva al sensore la vista del raggio infrarosso proveniente dall'uomo che comandava il robot.

Questo nuovo sistema rappresentava un'estensione del primo, non una sostituzione, infatti l'angolo di rotazione del servo motore era impostato a seconda di quale sensore ad infrarossi appartenente al primo sistema veniva eccitato.

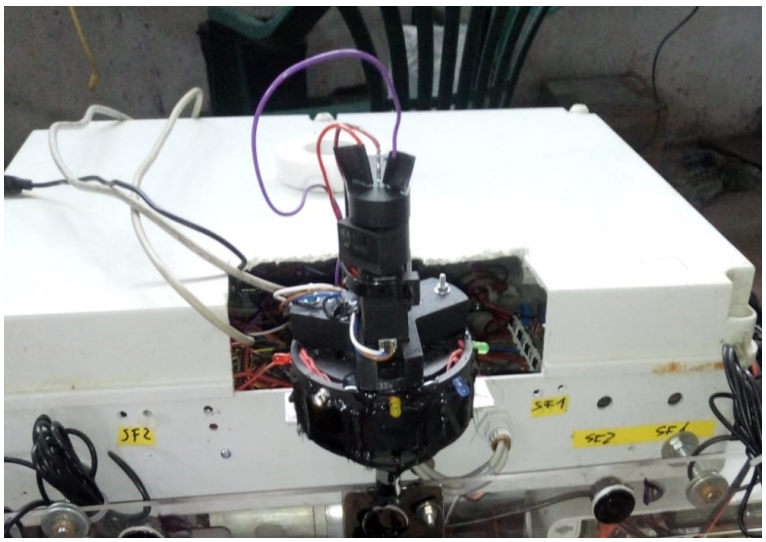


Figura 4.3: Fotografia del nuovo tipo di sensore IR posto in rotazione al di sopra dei già presenti sensori fissi

Il sistema risultava più preciso del primo riuscendo ad individuare l'operatore e a dire esattamente in quale angolo esso era posizionato, a differenza del primo che aveva un'approssimazione di diversi gradi rispetto alla realtà.

4.7 Migrazione del progetto su ROS

Dalle considerazioni emerse, osservando le problematiche della versione precedente, vennero prese alcune decisioni che diedero una svolta decisiva al progetto. Per prima cosa venne sostituito il Raspberry Pi con una macchina ad architettura x64 con una potenza di calcolo nettamente maggiore al single board computer. Venne introdotto un laser scanner bidimensionale che sarebbe poi stato affiancato da altri 2 laser in un secondo momento, per rimpiazzare tutti i sensori ad ultrasuoni ed essere utilizzato come supporto al sistema di navigazione.

Infine, il passo più importante dal punto di vista dello sviluppo: l'adozione di ROS e di tutta la sua architettura come fulcro software. Di questi cambiamenti, quello che più diede nuova linfa al progetto fu proprio quest'ultimo. Il middleware non venne installato solamente sul PC centrale ma anche sulle schede Arduino attraverso l'impiego delle librerie messe a disposizione dal pacchetto rosserial_arduino, con il quale è possibile utilizzare buona parte delle funzioni base di ROS direttamente sulla scheda. In questo modo si era raggiunto l'obiettivo di una comunicazione unificata con il supporto di invio e ricezione di messaggi in modo nativo senza l'obbligo di una conversione in stringa. Il sistema risultava stabile e semplice da gestire, la latenza era bassa a sufficienza da non creare problemi di scarsa reattività.

```
1  //librerie lato Arduino fornite da rosserial_arduino
2  #include <ros.h>
3  #include <std_msgs/Empty.h>
4
5  //libreria della classe
6  #include "MotorControllerTask.h"
7
8  //dichiarazione del NodeHandle di ROS
9  ros::NodeHandle nh;
10
11 /*
12 * dichiarazione dei subscriber iscritti rispettivamente ai topic
13 * "motor_left_pwm" e "motor_right_pwm"
14 */
15 ros::Subscriber<std_msgs::Int32> sub_motor_left("motor_left_pwm",
```

```
16 &setPwmValueMotorLeft );
17
18 ros::Subscriber<std_msgs::Int32> sub_motor_right("motor_right_pwm",
19 &setPwmValueMotorRight );
20
21 //metodo di inizializzazione del task
22 void MotorControllerTask::init(int period){
23   Task::init(period);
24
25   //Inizializzazione del nodo e dei subscriber
26   nh.initNode();
27   nh.subscribe(sub_motor_left);
28   nh.subscribe(sub_motor_right);
29 }
30
31 //metodo in loop del task
32 void MotorControllerTask::tick(){
33
34   //spin di ROS
35   nh.spinOnce();
36
37   ...
38 }
39
40 //Metodo callback del subscriber relativo al motore sinistro
41 void setPwmValueMotorLeft( const std_msgs::Int32& pwm_value){
42   motor_left->setPwm(pwmValue);
43   return;
44 }
45
46 //Metodo callback del subscriber relativo al motore destro
47 void setPwmValueMotorRight( const std_msgs::Int32& pwm_value){
48   motor_left->setPwm(pwmValue);
49   return;
50 }
```

Se fino ad ora erano stati necessari circa 2 anni di sviluppo, con l'utilizzo di ROS ed il know-how accumulato, in circa 2 mesi il sistema fu riportato alle stesse caratteristiche di prima ma con l'impiego di tecnologie totalmente rinnovate. L'operatore a questo punto veniva individuato incrociando i dati

provenienti dalla ricezione dei raggi infrarossi e delle distanze rilevate dal laser scanner. Una volta inquadrato l'uomo, il robot modulava la propria velocità e la direzione grazie ad un algoritmo PID affinato impiegando numerose ore di test. In modo da ampliare le funzionalità del robot e progredire sempre di più con gli obiettivi prefissati all'inizio del progetto, venne aggiunto al robot il piano di carico. Attraverso un giroscopio e ad un secondo algoritmo PID, il piano, rimaneva sempre in posizione orizzontale anche in condizioni di pendenza del terreno. Fu sicuramente un grande passo in avanti anche la possibilità di accedere finalmente a tool di analisi e debug come rviz, ros_bag e tutti quelli messi a disposizione da ROS senza più la necessità di svilupparne di personalizzati, risparmiando ore di lavoro e ottenendo risultati sicuramente migliori.

4.8 Analisi dei risultati e possibili sviluppi

Lo sviluppo del progetto terminò in uno stato in cui una parte degli obiettivi fissati all'inizio erano stati raggiunti, in particolare quelli più basilari. In particolare le funzioni sviluppate erano:

- inseguimento dell'operatore da parte del robot
- adattamento della velocità a quella dell'operatore
- evitare gli ostacoli e qualsiasi collisione con oggetti e persone
- livellamento automatico del piano di carico

Altri invece purtroppo non furono mai realizzati anche se sicuramente avrebbero reso più completo e versatile l'intero progetto.

Alcuni di questi obiettivi mancati furono:

- monitoraggio remoto del robot da grandi distanze
- implementazione di un GPS
- pianificazione di percorsi da eseguire in modo autonomo

Per quello che è stato effettivamente programmato, il risultato penso sia buono considerando il numero di individui che hanno lavorato al progetto, ma ancora decisamente migliorabile.

La modalità con la quale il robot insegue l'operatore, durante i test, si è dimostrata essere efficace in relazione alla sua semplicità, ma non sufficientemente precisa all'interno di scenari più impegnativi dove la precisione gioca un ruolo fondamentale. In una versione successiva sarebbe stato interessante convertire quel sistema di riconoscimento con uno basato su telemetry, come nel caso di studio simulato, affiancandolo ad un sistema di riconoscimento della persona, così facendo la precisione non sarebbe più stata un problema. L'idea a quel punto sarebbe stata quella di eliminare l'identificazione dell'operatore da parte del robot ed affidarsi quasi unicamente alla posizione condivisa dall'operatore ad ogni passo.

Un punto estremamente importante, che avrebbe dato un valore aggiunto al progetto, sarebbe sicuramente potuto essere la capacità di impostare percorsi da eseguire in modo autonomo direttamente da un'interfaccia di controllo sfruttando la navigazione GPS ed i sensori di rilevamento degli ostacoli, in modo da eseguire operazioni utili come per esempio il ritorno in base senza operatore o l'invio di materiali da un punto ad un altro di un cantiere o di un azienda agricola.

Conclusioni

Durante l'utilizzo di ROS, sia all'interno dell'esempio di utilizzo simulato che del caso di studio reale, ho potuto apprezzare vari aspetti peculiari di ROS.

I sensori e le schede Arduino vengono utilizzate attraverso ROS con grande semplicità grazie alle librerie, sviluppate direttamente dalla community, che ne garantiscono una piena compatibilità con il sistema, cosa non comune su altri framework che presentano a volte incompatibilità con hardware non ufficialmente supportato. Il funzionamento, inoltre, è risultato stabile e reattivo malgrado la scarsità di risorse.

L'apprendimento di ROS non è stato drammatico, non si può dire sia qualcosa di elementare, ma credo che con l'ausilio della documentazione e della wiki, qualsiasi sviluppatore sia in grado di iniziare a sviluppare su questa piattaforma senza un grosso impiego di ore, a patto di conoscere già discretamente C++ o Python.

L'installazione dei pacchetti provenienti da terze parti all'interno del software rasenta il banale. La forte modularità del sistema permette che un copia-incolla dell'intero sorgente del pacchetto all'interno della cartella relativa ai sorgenti del proprio progetto permetta di utilizzarlo ed implementarlo senza ulteriori sforzi, ovviamente previo rispetto delle dipendenze, soddisfabili in modo similare.

Diverso è il discorso per utilizzare ROS al pieno delle sue potenzialità, essendo così versatile, le possibilità di raffinamento degli algoritmi e la loro evoluzione ha veramente pochi limiti da parte del middleware. Tutto questo gioca sicuramente un ruolo fondamentale nella corsa all'automazione e allo sviluppo sempre più efficace di una robotica di servizio sempre più intelligente ed autonoma che, come evidenziato all'interno dell'introduzione, è in forte espansione.

Un esempio di ciò che ho appena descritto è riconducibile all'integrazione degli algoritmi di movimento del robot in modo autonomo. Durante la prima fase di implementazione mi è sembrato piuttosto semplice e "divertente" utilizzarli e notare come questi si integrassero perfettamente con il sistema e fossero utilizzabili con pochissimi sforzi, permettendo al robot di compiere i primi movimenti dopo pochissimi passi di sviluppo. Diverso il discorso relativo al successivo tuning del sistema e alla personalizzazione del comportamento di base: complesso trovare la combinazione di settaggi giusta per un movimento che fosse il più naturale possibile. Stesso discorso per ampliare o modificare le funzionalità offerte dai pacchetti distribuiti, non tanto per la tecnologia, quanto piuttosto per il livello di conoscenza necessaria per farlo.

Se ROS mostra i muscoli all'interno dello sviluppo di robot tramite la sua modularità, una suite di tool molto potenti ed estremamente efficienti per il debug e l'analisi, l'adozione di linguaggi di programmazione largamente diffusi come C++ e Python, non riesce però ad eccellere ed essere utilizzato agevolmente in altri campi dell'industria 4.0. Pensando a ROS all'interno dello sviluppo di tecnologie embedded o in ambito real-time, infatti, presenta non pochi limiti nel suo impiego. Partendo dal secondo, ROS è possibile installarlo su sistemi operativi GNU/Linux consumer come Ubuntu e Debian, dove il kernel non ha nessuna proprietà real-time. In secondo luogo il middleware utilizza come protocolli di comunicazione TCP e UDP, non propriamente adatti ad un utilizzo in situazioni critiche, proprio per la loro "scarsa" affidabilità e le performance non sufficientemente elevate per questo ambito. Ad onor del vero esistono alcuni *package*, come per esempio *pr2_controller_manager*, che sostituiscono alcune funzionalità base di ROS con altre pensate per un utilizzo real-time, ma non credo siano comunque sufficienti per affermare che sia adeguato per svolgere certe tipologie di operazioni. Parlando invece di tecnologie embedded pensate per la domotica, per quanto ROS sia relativamente semplice e la sua architettura a grafo vada incontro a diverse tipologie di utilizzo, vi sono però tantissimi framework molto più semplici in commercio specializzati proprio per questo utilizzo.

OpenHub, Node-RED e tanti altri utilizzano linguaggi molto più semplici, come per esempio Javascript e Lua, fornendo strumenti per lo sviluppo spesso visuali al programmatore, accompagnandoli con AI proprietarie specializzate,

come nel caso di Watson di IBM che è integrata all'interno di Node-RED. Questi framework, inoltre, solitamente utilizzano protocolli di comunicazione molto più leggeri e semplici, come l'MQTT, molto diffuso in questo tipo di progetti.

Ringraziamenti

Ci tengo a ringraziare in primo luogo il professor Ricci Alessandro, relatore di questa tesi, che mi ha guidato e consigliato in modo eccellente durante tutto l'intero percorso.
Un ringraziamento speciale va ai miei genitori e a mia sorella Eleonora, bussole essenziali durante il cammino che mi ha portato fin qui, senza di loro tutto questo non sarebbe mai stato possibile.

Ad Alessia, colei che mi ha supportato e sopportato in ogni istante, da Algebra fino alla laurea, senza di lei questi ultimi anni sarebbero stati del tutto diversi.

Grazie ad Edoardo, amico di una vita che ha saputo alleggerire e rendere decisamente divertente l'intero percorso.

Grazie a Nicola che nonostante tutto è rimasto sempre presente spronandomi ad andare avanti oltre tutte le difficoltà

Un grazie a Davide per il grande aiuto, persona chiave durante la preparazione di alcuni importanti esami e fonte di preziosi consigli.

Bibliografia

[1] IFR, "Service robots." https://ifr.org/service-robots.

[2] B. Group, "Exoskeleton vest in bmw group plant spartanburg (03/2017)." https://mediapool.bmwgroup.com/download/edown/pressclub/publicq?dokNo=P90249799&attachment=1&actEvent=image.

[3] Fendt, "Mars (mobile agricultural robot swarms)." https://www.fendt.com/int/fendt-mars.

[4] FENDT, "Unità robotizzate di piccole dimensioni funzionanti in gruppo." https://www.fendt.com/it/images/59968dbc994690e3138b4573_1503038909_web_it-IT.jpg.

[5] NASA, "Nasa robonaut 2." https://www.nasa.gov/images/content/421727main_jsc20093155300.jpg.

[6] NASA, "Towards autonomous operation of robonaut 2." https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20110024047.pdf.

[7] W. Garage, "Tourtlebot 2." https://www.turtlebot.com/assets/images/turtlebot_2_lg.png.

[8] P. S.p.a., "Modulo pilz ros." https://www.pilz.com/imagecache/mam/pilz/images/uploads/fittosize__752_0_ac18888d46a5284d670a750269e86e26_f_fts_dualarm_manipulator_sps_2018_mmerz_015_2019_05_09_v0-desktop-1565595259.jpg.

[9] P. S.p.a., "Pacchetti ros per la vostra applicazione robotica." https://www.pilz.com/it-IT/products-solutions/robotics/ros-modules.

[10] R. B. Rusu, "Esempio api ros master." https://wiki.ros.org/Events/CoTeSys-ROS-School?action=AttachFile&do=view&target=ros_tutorial.pdf.

[11] R. Wiki, "Packages." https://wiki.ros.org/Packages.

[12] R. Wiki, "Metapackages." https://wiki.ros.org/Metapackages.

[13] R. Wiki, "Message types." https://wiki.ros.org/msg.

[14] R. Wiki, "Service types." https://wiki.ros.org/srv.

[15] R. Wiki, "Nodes." https://wiki.ros.org/Nodes.

[16] R. Wiki, "Master." https://wiki.ros.org/Master.

[17] R. Wiki, "Parameter server." https://wiki.ros.org/ParameterServer.

[18] R. Wiki, "Messages." https://wiki.ros.org/Messages.

[19] R. Wiki, "Topics." https://wiki.ros.org/Topics.

[20] R. Wiki, "Services." https://wiki.ros.org/Services.

[21] R. Wiki, "Bags." https://wiki.ros.org/Bags.

[22] R. Wiki, "Ros documentation." https://wiki.ros.org/Documentation.

[23] J. F. Dave Hershberger, David Gossow, "Rviz." https://wiki.ros.org/rviz.

[24] R. Answers. answers.ros.org/upfiles/14075513525710936.png.

[25] T. F. Aaron Blasdel, "Rqt_bag." https://wiki.ros.org/rqt_bag.

[26] R. Wiki, "Rqt bag." https://wiki.ros.org/rqt_bag?action=AttachFile&do=get&target=rqt_bag.png.

[27] D. Thomas, "Rqt_graph." https://wiki.ros.org/rqt_graph.

[28] R. Wiki, "Rqt graph." https://wiki.ros.org/rqt_graph?action=AttachFile&do=get&target=snap_rqt_graph_moveit_demo.png.

[29] R. P. R. Joao Machado Santos, David Portugal, "An evaluation of 2d slam techniques availablein robot operating system." https://home.isr.uc.pt/~davidbsp/publications/SPR_SSRR2013_SLAM.pdf.

Elenco delle figure