UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA

Corso di Laurea triennale in INGEGNERIA GESTIONALE

Esplorazione e navigazione autonoma per robot umanoidi basata su ROS

Exploration and autonomous navigation for humanoid robots based on ROS

Relatore:
Prof. Andrea Monteriù

Tesi di laurea di:
Alfiero Frate

Correlatore:
Prof. Sabrina Iarlori

A.A 2019/2020

Ai miei genitori

Indice

Introduzione alla robotica

Lo sviluppo hardware nel campo della robotica ha raggiunto negli ultimi anni livelli impressionanti ed è in continua crescita. La robotica è la branca dell'ingegneria che si occupa della progettazione, dello sviluppo e del funzionamento dei robot, dei sistemi informatici per l'elaborazione dei dati provenienti dai sensori e del loro utilizzo per delineare il comportamento de tenere al fine di eseguire compiti specifici. Il termine fu coniato nel 1920 dallo scrittore-scienziato ceco Karel Capeke ha il significato di '' lavoro forzato'', e in effetti il fine ultimo di un robot è quello di riprodurre in qualche forma il lavoro umano. Un robot deve affrontare tutte le difficoltà provenienti dall'interazione col mondo reale.

Per poter affrontare questa sfida, in primis vengono richieste conoscenze dei modelli matematici e della fisica, soprattutto per quanto concerne tutta la parte di acquisizione e elaborazione dei dati, essenziale per potersi rapportare con l'ambiente in cui il robot interagisce.

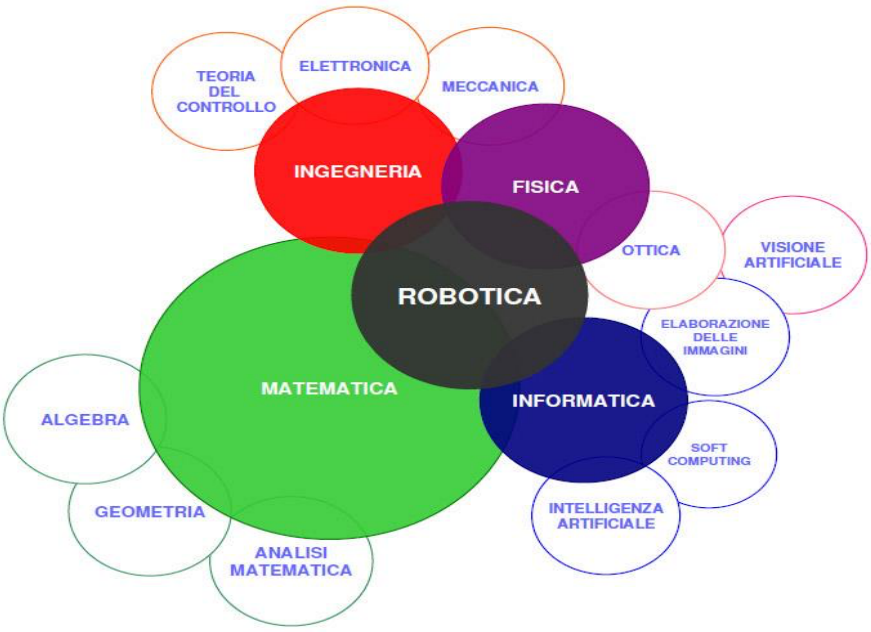

Figura 1-La robotica come scienza interdisciplinare.

L'informatica è certamentela disciplina cardine per tutta la parte computazionale, e in primo luogo per quello che riguarda il controllo.

Come deve reagire il robot ai dati sensoriali? Nel caso debba eseguire necessariamente più azioni, quale deve essere l'ordine?

Sono diverse le architetture di controllo sviluppate al fine di ricreare una sorta di ''intelligenza'' interna del robot, ed evidentemente il campo presenta una complessità non affrontabile dal singolo programmatore. Nel seguito della tesi verrà esposta una breve descrizione della struttura di un robot preso in studio, il Pepper, e si entrerà nel dettaglio del framework ROS come utile strumento nello sviluppo software e test di applicazione robotiche.

Cosa sono i Robot?

Nel 1979 l'istituto americano dei Robot fornisce una definizione di robot, ossia uno strumento programmabile e multifunzionale progettato per spostare materiali, componenti o attrezzi, attraverso vari movimenti programmati.

A vent'anni di distanza, questa definizione potrebbe essere considerata incompleta, dato che al giorno d'oggi un robot è visto come uno dispositivo altamente tecnologico utilizzato in campo scientifico e nell'industria per prendere il posto di un essere umano.

Caratteristiche di un robot:

- **Programmabilità**: capacità di elaborazione che il progettista può combinare come desidera;
- **Mobilità**: possibilità di interagire fisicamente con l'ambiente;
- **Flessibilità:** capacità di esibire un comportamento adatto alla situazione;
- **Autonomia:** possibilità di svolgere le proprie funzioni senza ingerenze o condizionamenti da parte di altri membri.

Un robot deve avere una struttura fisica con cui interagire col mondo esterno e affrontarne le sfide. Un robot che esiste solo all'interno di un computer è solo una simulazione che non affronta la vera complessità del mondo reale. Nella realtà, il robot ha la necessità di percezione dell'ambiente in cui si ritrova ad operare.

Pertanto, il robot deve essere dotato di sensori di vario tipo al fine di reperire informazioni riguardo il mondo circostante, al fine di poter rispondere in maniera adeguata agli ostacoli catturandone la presenza nel proprio percorso. Un robot deve quindi poter agire in risposta agli input derivanti dai sensori, e deve avere degli attuatori che gli permettono di compiere azioni specifiche nel mondo circostante.

In questo lavoro di tesi si approfondiscono alcune applicazioni robotiche, introducendo una piattaforma di sviluppo software che è la base su cui è costruita l'intera robotica moderna: ROS(Robotic Operating System). Verrà mostrato il suo funzionamento nel dettaglio, in particolare verranno descritti gli elementi fondamentali che lo rendono così importante nello sviluppo di applicativi. Successivamente verrà fornita una dettagliata descrizione del robot Pepper, oggetto del caso studio, ed in particolare verranno presentati gli algoritmi che permettono di navigare in sicurezza in un ambiente reale ed infine si concluderà con un semplice caso di studio in cui verrà messo in mostra concretamente l'utilizzo degli algoritmi di navigazione.

Per effettuare la navigazione autonoma il robot implementa una funzione di ricostruzione della mappa ambientale, permettendogli di ricreare la mappa dell'area dove il robot andrà ad operare; tale mappa viene quindi utilizzata per localizzarsi e per navigare nell'ambiente in totale sicurezza, evitando gli ostacoli circostanti. Importanti saranno sicuramente, come detto in precedenza, i dati provenienti dai sensori, ed in particolare dal sensore laser.

Il lavoro di tesi si articola in cinque capitoli di cui se ne fornisce di seguito una rapida descrizione:

- Capitolo 1: si presenta la piattaforma ROS con una breve infarinatura sulle caratteristiche generali;
- Capitolo 2: si descrive l'architettura ROS con i tre livelli concettuali;
- Capitolo 3: si descrivono brevemente algoritmi che permettono la navigazione autonoma di un robot umanoide e come sono implementati nella piattaforma ROS;

- Capitolo 4: si presenta il robot umanoide Pepper, caso di studio, con le funzionalità , caratteristiche principali e alcuni risultati raggiunti da questo lavoro;
- Capitolo 5: conclusioni e considerazioni;

Capitolo 1

ROS: Robot Operating System

1.1 *Introduzione a ROS*

Figura 2 : Logo di ROS. (fonte: http://www.ros.org)

Robot Operating System" (ROS) è un set di librerie software e strumenti pensati per agevolare la creazione di applicazioni per la robotica. È un progetto open-source, supportato dalla "Open Source Robotics Foundation", che può contare su una comunità di utenti molto attiva. Il software ROS è strutturato in modo modulare così da essere facilmente distribuito; oggi ROS può contare oltre 3000 pacchetti sviluppati. I componenti fondamentali di ROS (detti "core") sono distribuiti sotto licenza BSD, che ne permette l'utilizzo anche in prodotti di natura commerciale.

1.2 *Storia*

Lo sviluppo di ROS ebbe ufficialmente inizio nel 2007 sotto il nome di Switchyard presso lo Stanford Artificial Intelligence Laboratory a supporto del progetto Stanford Artificial Robot (STAIR) e del programma Personal Robotics (PR). Nel 2008 lo sviluppo del progetto progredisce presso il Willow Garage, ente di ricerca e incubatore nel campo della robotica che versa il proprio contributo approfondendo i principi con cui il progetto nasce e realizzando le prime versioni testate. Qua assume il nome ROS, Robot Operating System, nel 2010 viene rilasciata la prima versione stabile presentante la maggior parte delle caratteristiche odierne, e continua lo sviluppo fino al 2013 quando avviene la transizione presso l' Open Source Robotics Foundation e il Willow Garage viene assorbito dalla Clearpath Robotics. E' un software gratuito sia per l'ambito di ricerca che per fini commerciali. Ciò ha favorito fin da subito il suo sviluppo nel corso degli anni grazie al contributo della comunità di ricerca mondiale ed è divenuto uno dei suoi punti di forza.

Figura 3-Storia del framework ROS.

1.3 *Caratteristiche del ROS*

ROS è una collezione in continua crescita di librerie software open source e tools progettate al fine di aiutare gli sviluppatori nella realizzazione di applicazioni robot. Acronimo di Robot Operating System, a discapito del nome non si identifica in un sistema operativo nel senso stretto del nome. Viene definito infatti come meta-operating system, inglobando sì le caratteristiche tipiche di un vero e

proprio sistema operativo (astrazione dell'hardware sottostante, gestione dei processi, package managment, gestione dei dispositivi) ma arricchendolo con elementi tipici di un middleware (fornisce l'infrastruttura per la comunicazione tra processi/macchine differenti), e di un framework (tools di utilità per lo sviluppo, debugging e simulazione).

What Makes the Difference...?

Conventional OS	ROS
Explicitly a general purpose OS	Exclusive for Robotic Platform
Ortho-Operational	Meta-Operational
Native Language Programming	Language-independent architecture
Sequential Architecture	Asynchronous Distributed architecture
Programming IDE	Software Frameworks
Propriety/Open-Source	Open-source under BSD license
Heavily coded frameworks	ROS frameworks are very light
Programs	Nodes
Communication	Messages
Kernel is Included	Kernelless

ROS opera essenzialmente su piattaforme Unix-based, anche se sono innumerevoli le piattaforme sperimentali.

Figura 5-Sistemi operativi supportati da ROS.

ROS attualmente supporta tre differenti linguaggi: C++, Phyton, LISP.

Figura 6-Linguaggi attualmente supportati da ROS.

Inoltre vengono esposte di seguito alcune delle numerose piattaforme robotiche che supportano il framework ROS.

Figura 7-Piattaforme robotiche che supportano il framework ROS.

Capitolo 2

Architettura ROS

ROS è basato su un'architettura a grafo dove il processamento avviene nei nodi, che comunicano tra loro attraverso lo scambio di messaggi in maniera asincrona

attraverso l'uso di topic ai quali possono sottoscrivere e/o sui quali possono pubblicare, e in maniera sincrona con la chiamata di servizi, simili a RPC.

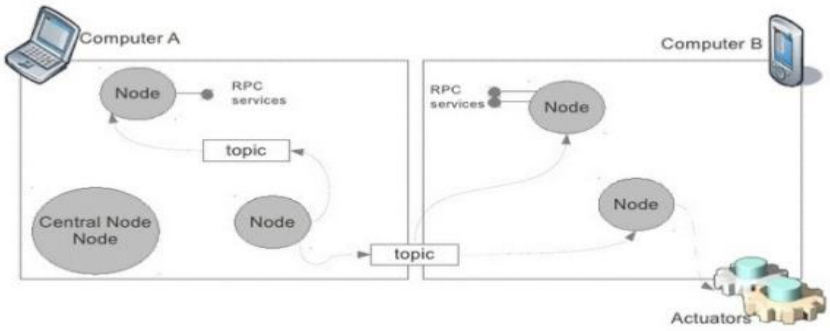

Figura 8-Esempio architettura ROS.

Strutturalmente ROS si sviluppa su tre livelli concettuali; Filesystem Level, Computational Level e Community Level, di cui andremo ad esaminare gli elementi costitutivi e il loro ruolo nell'architettura .

2.1.1 Filesystem Level

Figura 9- Primo livello concettuale; Filesystem.

Il Filesystem Level comprende tutte le risorse utilizzate in ROS, in particolare Packages, Metapackages, Package Manifest, Stacks, Message (msg) Type, Service (srv) Type.

2.1.1.1 Packages

In ROS il software viene organizzato in packages, che in sostanza costituiscono dei moduli. Ogni package può contenere un nodo ROS, un file di configurazione, un dataset, una libreria, software di terze parti etc. Lo scopo dei packages è ottenere una maniera efficace per il riuso del codice, venendo strutturati in modo tale da risultare funzionali ma non troppo pesanti e difficili da utilizzare.

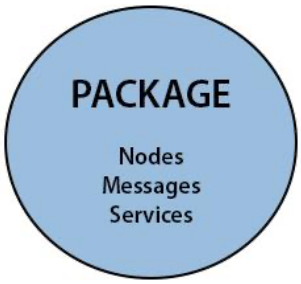

Figura 10- Packages.

2.1.1.2 Metapakages

Costituiscono forme particolari di package che non contengono nè file nè codice. Vengono usati per descrivere che più package sono legati assieme.

2.1.1.3 Stacks

Sono collezioni di packages con il comune scopo di svolgere un determinato task. Così a titolo di esempio il " Navigation Stack " è lo stack contenente tutti i package necessari al movimento del robot. Ogni Stack ha associata una versione e può dichiarare dipendenze da altri Stack. Sono in ROS il meccanismo primario per la distribuzione del software.

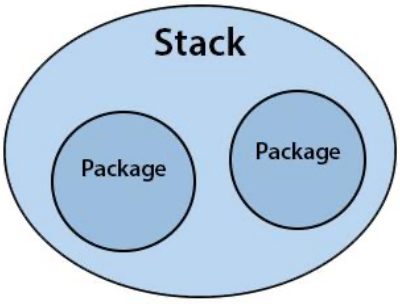

Figura 11- Stack.

2.1.1.4 Message (msg) Type

Indica il tipo di dato del messaggio, che determina di conseguenza il contenuto del messaggio stesso. È strutturato come una lista di campi, uno per linea, ognuno dei quali è definito dalla concatenazione di un tipo di dato built-in (come bool, string, float64 etc) e nome del campo.

2.1.1.5 Service (msg) Type

Indica il tipo di dato del messaggio scambiato nell'uso dei servizi di ROS, che determina di conseguenza il contenuto del messaggio stesso. Alla stessa maniera del message type, un service data type è definito da una collezione di campi con nome come visto in precedenza. Tuttavia la differenza consiste nel fatto che un service data type è suddiviso in due parti. Essendo utilizzati in un modello di comunicazione request/response, una parte dei campi sarà relativa alla request, mentre l'altra relativa alle response.

2.1.2 Computation Graph Level

Figura 12- Secondo livello concettuale; Computation Graph Level.

Il Computation Graph consiste nella rete peer-to-peer descritta da tutti i processi attivi in ROS che stanno elaborando dati assieme. L'utilizzo di una topologia peer-to-peer permette di evitare carichi di traffico di messaggi eccessivi quali si potrebbero avere tra i componenti periferici e ad esempio un server centrale, nel caso in cui i computer operanti siano collegati in una rete eterogenea, ma richiede un name service per rendere possibile ai nodi di contattarsi a vicenda nella rete. Verrano descritti in seguito gli elementi base del Computation Graph, nodi e master, Topic e servizi per quanto riguarda la comunicazione con scambio di messaggi rispettivamente in maniera asincrona e sincrona, e bag.

2.1.2.1 Nodi

Un nodo è un processo che compie una qualsiasi attività computazionale all'interno del sistema ROS. Essendo ROS progettato per essere modulare, in molte situazioni si ha un sistema basato su di esso comprende numerosi nodi, e in tal contesto sono interpretabili come moduli software, ognuno dei quali incaricato di gestire un aspetto del comportamento del robot, come ad esempio la parte decisionale, il movimento, l'azionamento dei motori etc. Un sistema il cui carico computazionale venga ripartito tra i vari nodi di cui è costituito ha innanzitutto il vantaggio di una maggiore tolleranza agli errori, potendo gestire il crash del singolo nodo. La complessità del codice è ridotta se confrontata coi sistemi monolitici e i dettagli implementativi sono

nascosti in quanti i singoli nodi offrono un'interfaccia composta da una API minimale. Ogni nodo in esecuzione dispone di quello che viene definito " graph resource name ", un nome che lo identifica unicamente al resto del sistema, e di un tipo che semplifica il processo di indirizzamento di un nodo eseguibile all'interno del filesystem. Tutti i tipi sono trattati come " package resource names", costituiti dalla concatenazione del nome del package e del file eseguibile del nodo.

2.1.2.2 Master

In un sistema basato su ROS, il Master è un server centralizzato XML-RPC che offre ai nodi un servizio di registrazione e di naming, similmente all'informazione data da un server DNS. Permette infatti al singolo nodo di contattarne un secondo attraverso una metodologia peer-to-peer. Tiene inoltre traccia per ogni singolo topic dei relativi pubblisher e subscriber allo stesso modo gestisce i servizi. Il Master offre anche il Parameter Server. Implementato attraverso XMLRPC, offre un servizio di memorizzazione e consultazione di parametri a runtime ai nodi che ne richiedono i servizi attraverso un' API di rete. Essendo globalmente accessibile, offre il vantaggio a tutti i tool di sviluppo di poter analizzare in qualsiasi momento la configurazione dello stato del sistema e modificarne i parametri se necessario.

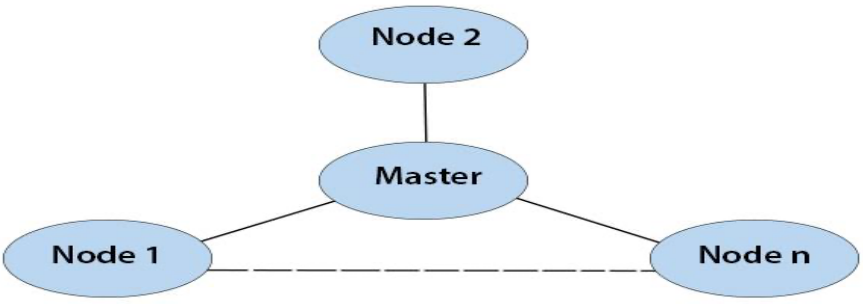

Figura 13- Server centralizzato; Master.

2.1.2.3 Messaggi

La comunicazione tra nodi avviene tramite scambio di messaggi. In ROS un messaggio è una struttura dati fortemente tipizzata. Sono supportati tutti i tipi standard primitivi (integer, floating point, boolean ecc.) così come array di tipi primitivi e costanti. Ogni messaggio può essere composto da altri messaggi o array di altri messaggi, arbitrariamente nidificati in complessità. La struttura di un messaggio è descritta da un semplice file di testo denominato msg file, di estensione .msg, contenuti nelle sottocartelle dei package. Un msg file è costituito da due parti : campi (fields) e costanti (constants). I campi sono i dati spediti all'interno del messaggio, mentre le costanti sono valori numerici utili a interpretare il significato dei campi.

2.1.2.4 Topic

I topic costituiscono il mezzo di comunicazione asincrono, unidirezionale, per lo scambio di messaggi tra nodi, secondo una semantica di tipo publish/subscribe. Ci possono essere più publisher e subscriber concorrenti per un singolo topic, e un singolo nodo può pubblicare e/o sottoscriversi a più topics. In generale, publisher e subscriber non sono consapevoli dell'esistenza degli altri, disaccoppiando cosi la produzione dell'informazione dal suo consumo. Ogni topic è fortemente tipizzato dal tipo di messaggio che viene pubblicato, e i nodi possono ricevere solo messaggi il cui tipo faccia matching. Ciò determina che all'interno del topic sia possibile scrivere o leggere un solo tipo di messaggio. Ogni nodo presenta un URI, che corrisponde alla concatenazione host:porta del server XMLRPC che è attivo. Tale server non

trasporta topic o dati, ma viene utilizzato per negoziare la connessione tra nodi e comunicare col Master. Il master ha altresì un well-known URI accessibile da tutti i nodi. La procedura sequenziale che permette a due nodi della rete di poter scambiare messaggi attraverso un topic si riassume nei seguenti passaggi:

1. Il publisher si registra al Master attraverso XMLRPC. Invia informazioni riguardo a ciò che andrà a pubblicare, tipo dei messaggi, nome del topic e il proprio URI.
2. Il subscriber si registra al Master attraverso XMLRPC. Anche qui vengono inviati informazioni su tipo dei messaggi, nome del topic e proprio URI. Il Master ritorna la lista di tutti gli URI degli attuali publisher, e aggiornamenti di quest'ultima qualora cambiasse.

Il Master mantiene una tabella in cui risiedono tutti i dati sia per i subscriber che per i publisher.

3. Il Master informa il subscriber del nuovo publisher tramite XMLRPC, comunicandogli l'URI.
4. Il subscriber contatta il publisher per richiedere la connessione al topic e negoziare il protocollo di trasporto, sempre tramite XMLRPC. TCP (TCPROS) è largamento usato per via della maggior affidabilità del suo stream . I pacchetti TCP giungono sempre in ordine, e quelli persi rimandati. Ma esistono anche casi in cui il protocollo UDP (UDPROS) risulata maggiormente efficace, ad esempio per comunicare simultaneamente in

maniera broadcast. ROS lascia la scelta del protocollo libera in fase di negoziazione.

5. Il publisher invia al subscriber la configurazione adatta al tipo di protocollo di trasporto scelto. Nel caso TCP ad esempio indirizzo IP e numero di porta.
6. Il subscriber si connette separatamente al publisher secondo il protocollo scelto.
7. I successivi messaggi pubblicati dal publisher verranno inviati al subscriber attraverso il canale aperto. Stabilita la connessione non è più richiesto il coinvolgimento del Master che in nessun momento è coinvolto nel flusso di dati scambiati nel topic.

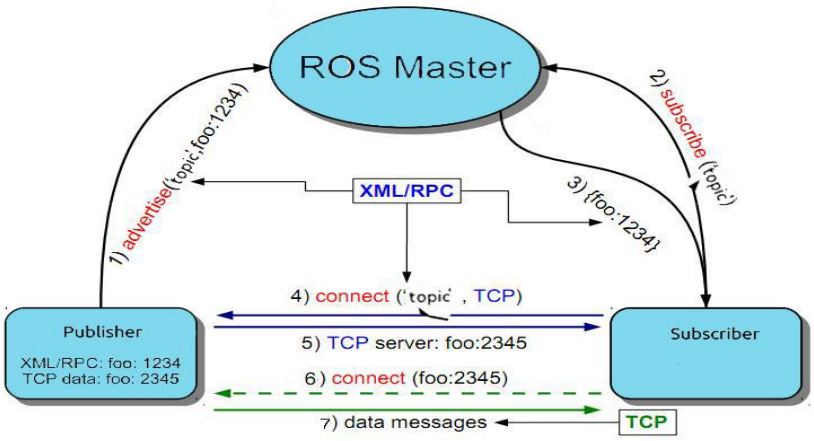

Figura 14-Esempio del mezzo di comunicazione Topic.

2.1.2.5 Servizi

I servizi rappresentano il mezzo di comunicazione sincrono secondo una semantica di tipo request / reply, implementando in ROS funzionalità di RPC. Andranno così a definirsi nella rete nodi che svolgeranno funzione di service provider e nodi client. Esistono funzioni per verificare la presenza di un service provider nella rete . Il nodo

client invierà dei dati che prenderanno il nome di request a un nodo server, aspettando la risposta di quest'ultimo. Il nodo server una volta ricevuta, processerà la request del client a cui inviere come risposta dei dati col nome di response. Un nodo client che voglia usufruire di un servizio inizializza un oggetto di tipo ros::ServiceClient che ha il compito di inoltrare la chiamata, e la cui signature è

```
ros::ServiceClient client = node_handle.serviceClient<service_type>(
service_name);
```

dove:

- Service_type è il service data type utilizzato nella comunicazione;
- Service_name è il nome passato come stringa del servizio che si vuole chiamare;

Successivamente si creano gli oggetti Request e Response :

```
package_name::service_type::Request package_name::service_type::Response
```

ed essendo lato Client riempita la Request coi dati da inviare al nodo server.

La chiamata al servizio:

```
bool success = service_client.call(request, response);
```

effettua infine il lavoro di localizzare il nodo server, trasmettere la Request, attendere l'arrivo della risposta e magazzinarne i dati ricevuti nei campi dell'oggetto Response creato in precedenza. Il ritorno è di tipo booleano indicandoci se la chiamata ha avuto

successo o meno. Lato server, come accadeva per i subscriber, ogni servizio che il nodo offre deve essere associato a una funzione di callback con signature

```
bool function_name( package_name::service_type::Request &req),
package_name::service_type::Response &resp) ){
...
}
```

Ros esegue la funzione di callback relativa una volta per ogni chiamata al servizio che il nodo riceve, il cui compito è essenzialmente riempire i campi dell'oggetto Response.

Successivamente si associa la funzione di callback col nome del servizio attraverso

```
ros::ServiceServer server = node_handle.advertiseService( service_name,
pointer_to_callback_function
);
```

dove:

- Service_name è il nome in formato stringa del servizio che si va ad offrire.
- Pointer_to_callback_function è il puntatore alla callback prima definita

Come visto coi topic, anche qui ROS non eseguirà nessuna callback finchè non sarà esplicitato con ros::spin() o ros::spinOnce():

Il protocollo di accesso a un servizio può essere riassunto nei seguenti passaggi:

1. Registrazione del servizio nel Master.

2. Richiesta di un determinato servizio al Master da parte di un client.

3. Creazione di una connessione TCP/IP al servizio da parte del Client.
4. Scambio tra Client e servizio di un Connection Header, il quale contiene importanti metadata sulla connessione che andrà a crearsi.
5. Invio del messaggio di request serializzato da parte del Client.
6. Invio del messaggio di response serializzato da parte del servizio.

Di default, le connessioni a un servizio sono stateless, e per ogni chiamata il Client desideri fare va ripetuto il protocollo sopra citato per ottenere una nuova connessione al service. Ciò permette un approccio più robusto e che un nodo che svolga la funzione di service provider possa venire riavviato, col tradeoff di un maggiore overhead. ROS tuttavia presenta anche una forma di connessione persistente a un service in cui viene mantenuta aperta al fine di permettere al client

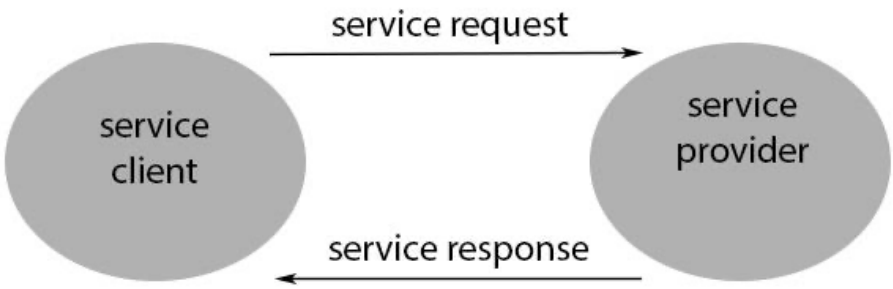

Figura 15- Schema di una richiesta di un determinato servizio al Master da parte di un client.

ripetuti invii di request. A un maggiore throughput, il tradeoff da pagare consiste nel fatto che nel caso in cui appaia un nuovo nodo provider del servizio a sostituzione del precedente, la connessione non verrà ugualmente interrotta, mentre se la connessione dovesse cadere per motivi tecnici non ci sarebbero tentativi di ripristino.

2.1.2.6 Bag

E' un formato file, con estensione .bag, utilizzato in ROS per la memorizzazione di dati di tipo ROS message. Viene creato dal tool rosbag, il quale si iscrive a uno o più topic e memorizza in un file i messaggi in forma serializzata. È il meccanismo sfruttato da ROS per fare il logging nelle comunicazioni topic.

2.1.3 Community level

Figura 16-Terzo livello computazionale; Community level.

Comprende tutte le risorse che permettono a comunità separate di ricercatori di poter interagire scambiando software. Sono collezioni di stacks versionati pronti all'installazione. Fino ad ora sono state tredici le distribuzioni rilasciate.

2.1.3.1 ROS wiki

È il forum principale per documentarsi su ROS. Dopo una veloce registrazione, si può contribuire alla crescita della comunità scrivendo tutorial, offrendo aiuto, correzioni e così via.

Capitolo 3

Navigazione autonoma di un robot umanoide

Come sopracitato, la navigazione autonoma di un robot umanoide è il perno di questo lavoro. È la capacità di spostarsi da un luogo ad un altro arbitrariamente costante ed è una delle competenze più impegnative richieste da un robot mobile.

Si deve fare riferimento ai quattro elementi principali della navigazione:

- Percezione: il robot infatti deve interpretare i suoi sensori per estrarre dati significativi;
- Localizzazione: il robot deve determinare la sua posizione nell'ambiente;
- Cognizione: il robot deve decidere come agire per raggiungere i suoi obiettivi;
- Controllo del movimento: il robot deve modulare le sue uscite per raggiungere la traiettoria desiderata;

  Una volta che le mappe ambientali sono disponibili, il robot deve pianificare nel modo più ottimale possibile per raggiungere i suoi obiettivi evitando gli ostacoli. Di seguito verranno riportati algoritmi per la navigazione mobile robotizzata.

3.1 Algoritmi per la navigazione e esplorazione autonoma

3.1.1 SLAM

Nella geometria computazionale e nella robotica, **la localizzazione e mappatura simultanea (SLAM)** è il problema computazionale della costruzione o dell'aggiornamento di una mappa di un ambiente sconosciuto mentre si tiene contemporaneamente traccia della posizione di un agente al suo interno. Il problema di SLAM è solitamente considerato come uno dei più importanti. Fortunatamente, questo algoritmo è già implementato come pacchetto ROS chiamato **Gmapping**, avente come nome **slam_gmapping**. Il requisito di base per far funzionare questo pacchetto è un robot che fornisce dati odometrici e ha un mirino laser orizzontale fisso. Utilizzando slam_gmapping, è possibile creare una mappa a griglia di occupazione 2D (come ad esempio la pianta di un edificio), tenterà inoltre di trasformare ogni scansione in entrata nel frame **tf** odom (odometria). (tf= trasformazioni necessarie per mettere in relazione i telai per laser, base e odometria)

Per creare una mappa da un robot con scansioni di pubblicazione laser sull'argomento base_scan:

rosrun gmapping slam_gmapping scan: = base scan

Il nodo slam_gmapping accetta i messaggi **sensor_msg / Laser scan** (Scansione laser da cui creare la mappa) e crea successivamente la mappa. Tale mappa può essere recuperata tramite un argomento o servizio ROS.

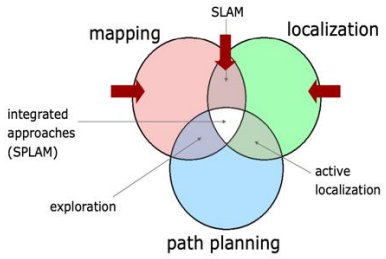

Figura 17- Algoritmo SLAM.

La performance dello SLAM diventa ancora peggiore se si utilizzano sensori a basso costo.

Per implementare questo algoritmo sulla piattaforma ROS abbiamo i seguenti **argomenti sottoscritti**:

tf(tf/tfMessage)

Trasformazioni necessarie per mettere in relazione i telai per laser, base e odometria.

scansione(sensor_msgs/LaserScan)

Scansioni laser da cui creare la mappa.

Argomenti pubblicati

map_metadata(nav_msgs/MapMetaData)

Si ottengono i dati della mappa da questo argomento, che è bloccato e aggiornato periodicamente.

map(nav_msg/OccupancyGrid)

Si ottengono i dati della mappa da questo argomento, che è bloccato e aggiornato periodicamente.

entropia(std_msgs/Float64)

Stima dell'entropia della distribuzione sulla posa del robot (un valore più alto indica una maggiore incertezza .

Servizi

dynamic_map (nav_msgs/GetMap)

Chiama questo servizio per ottenere i dati della mappa.

Parametri

inverted_laser (stringa , predefinito: "false")

laser è rivolto verso l'alto (le scansioni sono ordinate in senso antiorario) o capovolto (le scansioni sono ordinate in senso orario).

base_frame

Il telaio fissato alla base mobile.

map_frame

La cornice allegata alla mappa.

odom_frame

Il telaio attaccato al sistema di odometria.

map_update_interval

Quanto tempo (in secondi) tra gli aggiornamenti alla mappa. L'abbassamento di questo numero aggiorna la griglia di occupazione più spesso, a scapito di un maggiore carico computazionale.

maxUrange

La massima portata utilizzabile del laser. Un raggio viene ritagliato a questo valore.

kernelSize

Il kernel in cui cercare una corrispondenza.

lstep

La fase di ottimizzazione nella traduzione.

astep

La fase di ottimizzazione a rotazione

iterazioni

Il numero di iterazioni dello scanmatcher

lsigma

Il sigma di un raggio utilizzato per il calcolo della verosimiglianza

ogain

Guadagno da utilizzare durante la valutazione della probabilità, per smussare gli effetti di ricampionamento

skip

Numero di raggi da saltare in ogni scansione.

punteggio minimo

Punteggio minimo per considerare buono il risultato della scansione. Può evitare di saltare le stime di posa in ampi spazi aperti quando si utilizzano scanner laser con portata limitata.

srr, srt, str e stt

Errore di odometria nella traduzione.

linearUpdate

Elabora una scansione ogni volta che il robot trasla.

angularUpdate

Elabora una scansione ogni volta che il robot ruota.

temporalUpdate

Elabora una scansione se l'ultima scansione elaborata è precedente al tempo di aggiornamento in secondi. Un valore inferiore a zero disattiverà gli aggiornamenti basati sul tempo.

resampleThreshold

La soglia di ricampionamento basata su Neff

particelle

Numero di particelle nel filtro

xmin

Dimensione iniziale della mappa (in metri)

ymin

Dimensione iniziale della mappa (in metri)

xmax

Dimensione iniziale della mappa (in metri)

ymax

Dimensione iniziale della mappa (in metri)

llsamplerange e llsamplestep

Intervallo di campionamento traslazionale per la probabilità

lasamplerange e lasamplestep

Intervallo di campionamento angolare per la probabilità

occ_thresh

Soglia sui valori di occupazione di gmapping. Le celle con una maggiore occupazione sono considerate occupate.

maxRange

La portata massima del sensore.

3.1.2 AMCL

Un importante algoritmo di localizzazione è l'AMCL (Adaptive Monte Carlo Localization), ha una precisione di posa limitata a causa della non convessità del modello del sensore laser, delle caratteristiche complesse e non strutturate dell'ambiente di lavoro, della casualità del campionamento delle particelle e del problema di sezione della posa finale. AMCL quindi è un sistema di localizzazione probabilistico e spesso viene utilizzato nella navigazione. L'algoritmo si basa su un filtro di particelle che, data la mappa dell'ambiente, stima la posizione e l'orientamento di un robot mentre si muove e percepisce l'ambiente circostante.

Utilizza un filtro di particelle per rappresentare la distribuzione degli stati probabili, con ogni particella che rappresenta uno stato possibile, cioè un'ipotesi di dove si trova il robot. L'algoritmo inizia solitamente con una distribuzione uniforme e casuale delle particelle nello spazio di configurazione, il che significa che il robot non ha informazioni su dove si trova e suppone che sia ugualmente probabile che si trovi in qualsiasi punto dello spazio. Ogni volta che il robot si muove, sposta le particelle per prevedere il suo nuovo stato dopo il movimento e ogni volta che verrà percepito qualcosa, le particelle verranno ricampionate in base alla ricorsione. Questo nodo in ROS funziona solo con scansioni laser e mappe costruite con il laser.

Per localizzare utilizzando i dati laser sull'argomento base_scan:

scansione amcl: = base_scan

AMCL dunque acquisisce una mappa basata sul laser, esegue scansioni laser e trasforma i messaggi e fornisce stime di posa. Bisogna però stare molto attenti perché a causa delle impostazioni predefinite, se non è stato impostato alcun parametro, lo

stato iniziale sarà una nuvola di particelle di dimensioni moderate centrata su (0,0,0 rispettivamente in (X,Y,Z).

Per implementare questo algoritmo sulla piattaforma ROS abbiamo i seguenti **argomenti sottoscritti**:

scansione(sensor_msgs/LaserScan)

Scansione laser.

tf (tf/ tfMessage)

Trasforma.

Initialpose(geometry_msgs/PoseWithCovarianceStamped)

Media e covarianza con cui (ri) inizializzare il filtro di particelle

map(nav_msgs/ OccupancyGrid)

Quando il parametro use_map_topic è impostato, AMCL si abbona a questo argomento per recuperare la mappa utilizzata per la localizzazione basata sul laser.

Argomenti pubblicati

amcl_pose(geometry_msgs/PoseWithCovarianceStamped)

Posa stimata del robot nella mappa, con covarianza.

particlecloud (geometry_msgs/PoseArray)

L’insieme delle stime di posa viene mantenuto dal filtro.

tf(tf/tfMessage)

Pubblica la trasformazione da odom per mappare.

Servizi

Global_localization (std_srvs/Empty)

Avvia la localizzazione globale, in cui tutte le particelle vengono disperse in modo casuale attraverso lo spazio libero nella mappa.

request_nomotion_update(std_srvs/Empty)

Servizio per eseguire manualmente l’aggiornamento e pubblicare le particelle aggiornate.

set_map(nav_msgs/SetMap)

Servizio per impostare manualmente una nuova mappa e posa.

Servizi chiamati

static_map (nav_msgs/GetMap)

AMCL chiama questo servizio per recuperare la mappa utilizzata per la localizzazione basata sul laser; blocchi di avvio durante l’acquisizione della mappa da questo servizio.

Per quanto concerne i parametri, esistono tre categorie principali in ROS che possono essere dunque usati per configurare il nodo AMCL: filtro generale, modello laser e modello odonometrico.

Parametri di filtro generali

min_particles

Numero minimo consentito di particelle.

max_particles

Numero massimo consentito di particelle.

kld_err

Errore massimo tra la distribuzione vera e la distribuzione stimata.

kld_z

Quantile normale standard superiore per (1 - p), dove p è la probabilità che l'errore sulla distrubizione stimata sia inferiore a kld_err .

update_min_d

Movimento di traslazione richiesto prima di eseguire un aggiornamento del filtro.

update_min_a

Movimento rotatorio richiesto prima di eseguire un aggiornamento del filtro.

resample_interval

Numero di aggiornamenti del filtro richiesti prima del ricampionamento.

transform_tolerance

Tempo con cui postdatare la trasformazione pubblicata, per indicare che questa trasformazione è valida per il futuro.

recovery_alpha_slow

Tasso di decadimento esponenziale per il filtro a peso medio lento, utilizzato per decidere quando recuperare aggiungendo pose casuali. **Un buon valore potrebbe essere 0,001.**

recovery_alpha_fast

Tasso di decadimento esponenziale per il filtro del peso medio veloce, utilizzato per decidere quando recuperare aggiungendo pose casuali. Un buon valore potrebbe essere 0,1.

initial_pose_x

Media della posa iniziale (x), utilizzata per inizializzare il filtro con distribuzione gaussiana.

initial_pose_y

Media della posa iniziale (y), utilizzata per inizializzare il filtro con distribuzione gaussiana.

initial_pose_a

Media della posa iniziale (imbardata), utilizzata per inizializzare il filtro con distribuzione gaussiana.

initial_cov_xx

Covarianza della posa iniziale (x * x), utilizzata per inizializzare il filtro con distribuzione gaussiana.

initial_cov_yy

Covarianza della posa iniziale (y * y), utilizzata per inizializzare il filtro con distribuzione gaussiana.

initial_cov_aa

Covarianza della posa iniziale (imbardata), utilizzata per inizializzare il filtro con distribuzione gaussiana.

gui_publish_rate

Frequenza massima (Hz) alla quale le scansioni e i percorsi vengono pubblicati per la visualizzazione.

save_pose_rate

Frequenza massima (Hz) alla quale memorizzare l'ultima posa e covarianza stimata nel server dei parametri, nelle variabili ~ initial_pose_ * e ~ initial_cov_ *. Questa posa salvata verrà utilizzata nelle esecuzioni successive per inizializzare il filtro.

use_map_topic

Se impostato su true, AMCL si iscriverà all'argomento della mappa invece di effettuare una chiamata di servizio per ricevere la sua mappa.

first_map_only

Se impostato su true, AMCL utilizzerà solo la prima mappa a cui si iscrive, invece di aggiornarsi ogni volta che ne viene ricevuta una nuova.

selettivo_resampling

Se impostato su true, ridurrà la velocità di ricampionamento quando non è necessario e aiuterà a evitare la privazione delle particelle.

laser_min_range

Intervallo di scansione minimo da considerare.

laser_max_range

Intervallo di scansione massimo da considerare.

laser_max_beams

Quanti raggi equidistanti in ciascuna scansione da utilizzare durante l'aggiornamento del filtro.

laser_z_hit , short, max e rand

Peso della miscela per la parte z_hit, short, max e rand del modello.

laser_sigma_hit

Deviazione standard per il modello gaussiano utilizzato nella parte z_hit del modello.

laser_lambda_short

Parametro di decadimento esponenziale per la parte z_short del modello.

laser_likelihood_max_dist

Distanza massima per eseguire l'inflazione degli ostacoli sulla mappa, da utilizzare nel modello likelihood_field.

Parametri del modello laser

laser_min_range

Intervallo di scansione minimo da considerare; -1,0 farà sì che venga utilizzato il raggio minimo riportato del laser.

laser_max_range

Intervallo di scansione massimo da considerare; -1,0 farà sì che venga utilizzata la portata massima riportata del laser.

laser_max_beams

Quanti raggi equidistanti in ciascuna scansione da utilizzare durante l'aggiornamento del filtro.

laser_z_hit , short, max e rand

Peso della miscela per la parte z_hit, short, max e rand del modello.

laser_sigma_hit

Deviazione standard per il modello gaussiano utilizzato nella parte z_hit del modello.

laser_lambda_short

Parametro di decadimento esponenziale per la parte z_short del modello.

laser_likelihood_max_dist

Distanza massima per eseguire l'inflazione degli ostacoli sulla mappa, da utilizzare nel modello likelihood_field.

Parametri del modello odometrico

odom_model_type

Quale modello utilizzare, "diff" , "omni" , "diff-corretto" o "omni-corretto" .

odom_alpha1

Specifica il rumore previsto nella stima della rotazione dell'odometria dalla componente rotazionale del movimento del robot.

odom_alpha2

Specifica il rumore previsto nella stima della rotazione dell'odometria dalla componente traslazionale del movimento del robot.

odom_alpha3

Specifica il rumore previsto nella stima della traduzione dell'odometria dalla componente traslazionale del movimento del robot.

odom_alpha4

Specifica il rumore atteso nella stima della traslazione dell'odometria dalla componente rotazionale del movimento del robot.

odom_alpha5

Parametro di rumore relativo alla traduzione (utilizzato solo se il modello è "omni").

odom_frame_id

Quale cornice usare per l'odometria.

base_frame_id

Quale telaio utilizzare per la base del robot.

global_frame_id

Il nome del frame di coordinate pubblicato dal sistema di localizzazione.

tf_broadcast

Impostalo su false per impedire ad amcl di pubblicare la trasformazione tra il frame globale e il frame dell'odometria.

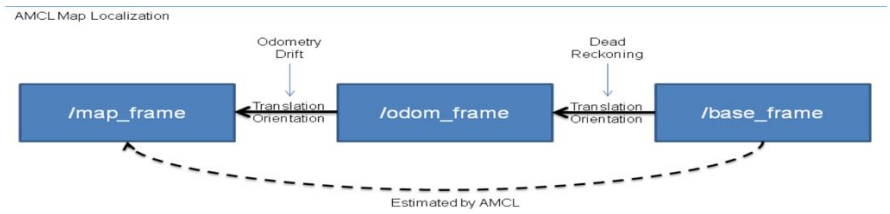

Figura 18- Mappa di localizzazione AMCL.

Questa figura rappresenta il funzionamento della mappa di localizzazione AMCL.

3.1.3 Odometria

L’Odometria è la tecnica per stimare la posizione di un veicolo su ruote che si basa su informazioni provenienti da sensori che misurano lo spazio percorso da alcune delle ruote e l’angolo di sterzo.

Utilizza dunque gli enconder incrementali montati su attuatori, che azionano e misurano le variazioni della loro rotazione, richiedendo che la posizione del robot sia riferita ad un sistema di coordinate definito. L’errore che viene commesso si accumula nel tempo con la distanza percorsa, infatti ha una scarsa precisione sulle lunghe distanze.

In ROS l’odometria utilizza il messaggio nav_msgs e con una serie di trasformazioi si passa da un frame di coordinate ‘’odom’’ a un frame di coordinate ‘’base_link’’ su tf. Tf in questo caso viene usato per determinare la posizione del robot nel mondo e correlare i dati del sensore a una mappa statica. Per giunta tf non fornisce alcuna informazione sulla velocità del robot ecco perché lo stack di navigazione richiede che qualsiasi sorgente di odometria pubblichi sia un messaggio di trasformazione che un messaggio nav_msgs/ Odometry che contenga in più informazioni riguardanti la velocità.

In conclusione in questo capitolo si è sottolineato come i pacchetti di importanza centrale per i compiti di mappatura e navigazione, ovvero gmapping e amcl, siano predisposti per funzionare con messaggi sensor_msgs/Laserscan, ottenuti dal sensore montato sul robot. L’utilizzo di tale dispositivo permette di ottenere informazioni di tipo esclusivamente bidimensionale sugli ostacoli nelle vicinanze, in quanto restituisce tutti e soli i punti in cui un ben precisato piano interseca gli oggetti all’interno del sensore.

Capitolo 4

Caso di studio: il robot umanoide Pepper

In questo capitolo andrò ad approfondire il mio oggetto di studio, il robot umanoide Pepper. La robotica umanoide è in assoluto tra i campi di ricerca più affascinanti. Il suo obiettivo è realizzare robot dalle sembianze umane, dotati di intelligenza artificiale e in grado di agire autonomamente. La nazione guida di questo settore è il Giappone, dove da una quindicina di anni si lavora al robot umanoide più avanzate al mondo. Si chiama **Asimo** ed è stato realizzato dalla Honda: servomotori, sensori e videocamere gli consentono di replicare i nostri movimenti e di rispondere ai comandi vocali.

Robot umanoidi come Asimo o Pepper sono stati progettati per essere utilizzati prevalentemente in ambiente domestico, ma ne esistono anche altri con finalità educative come Nao.

4.1 Caratteristiche e descrizione generale

Pepper come detto nel paragrafo precedente è un robot umanoide progettato da Softbank Robotics. Originariamente creato sotto la società francese Aldebaran Robotics, ampiamente conosciuta per il suo robot altamente interattivo NAO, è stato acquistato da Softbank Mobile Group nel 2013.

Il concetto di Pepper mostra il desiderio di produrre un robot amichevole compagno di robot che sia accessibile ad una più ampia gamma di clienti. Le sue caratteristiche principali generano la capacità di comprendere le emozioni umane e di reagire in modo predefinito a tali emozioni. Dal punto di vista tecnico, il robot incorpora una serie di sensori di prossimità e di visione che consentono lo sviluppo di algoritmi di tracciamento, localizzazione e navigazione. Inoltre presenta un sistema di guida a

ruota olonomico, consentendo una gamma più ampia di movimenti che si adatta adeguatamente agli scenari umani. Naoqi è il nome del software principale che gira sul robot e lo controlla. Attraverso l'uso di questo software, qualsiasi sviluppatore può creare una vasta gamma di applicazioni con Pepper. I linguaggi supportati sono Python e C++. Riguardo alle caratteristiche,questo robot pesa circa 29kg, con la maggior parte di questo distribuito nella parte inferiore del corpo. Questo particolare, ottimizza il movimento della parte superiore del corpo consentendo una maggiore gamma di movimenti nelle braccia e nel busto, con un totale di 20 gradi di libertà; 17 nel corpo e 3 nella base. Nella figura sottostante sono rappresentate le dimensioni principali (a sinistra) e le parti del corpo a destra.

Presenta inoltre LED e un tablet che hanno la funzionalità di migliorare la capacità del robot di comunicare, fornire feedback all'utente e rendere più trasparente lo stato del robot. In particolare gli occhi LED sono utilizzati per mostrare quando sta ascoltando. L'uso del tablet sul petto è triplice:

- Come dispositivo di didascalia;
- Schermo che mostra tutto ciò che dice il robot;
- Come dispositivo di input se l'utente ha bisogno di digitare qualcosa;

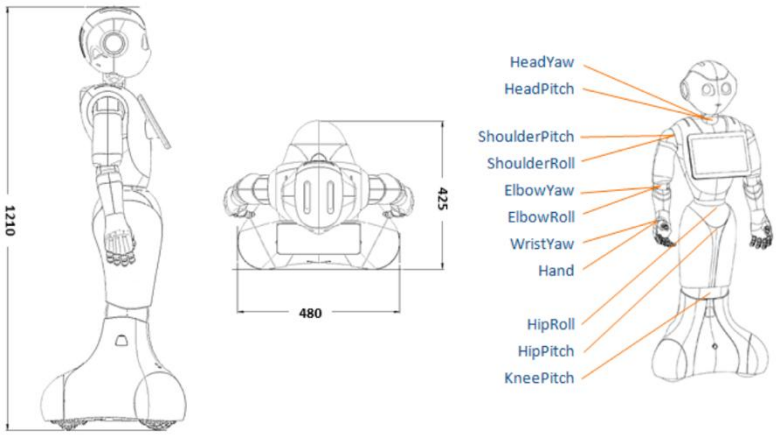

Figura 19-Dimensioni principali del Pepper (a sinistra), parti del corpo (a destra).

A Treviso è avvenuto il debutto di questo robot, il progetto dell'Università Cà Foscari di Venezia e, per la parte tecnica, della Promoservice di Chiarano (Treviso) che ha sottoposto Pepper, di lingua madre italiana, ad un corso in inglese e tedesco.

4.2 Sensori

Pepper ha due sonar costruiti nella parte anteriore e posteriore del KneePitch.

Fornisce il rilevamento di oggetti tra 0.3 e 5 metri.

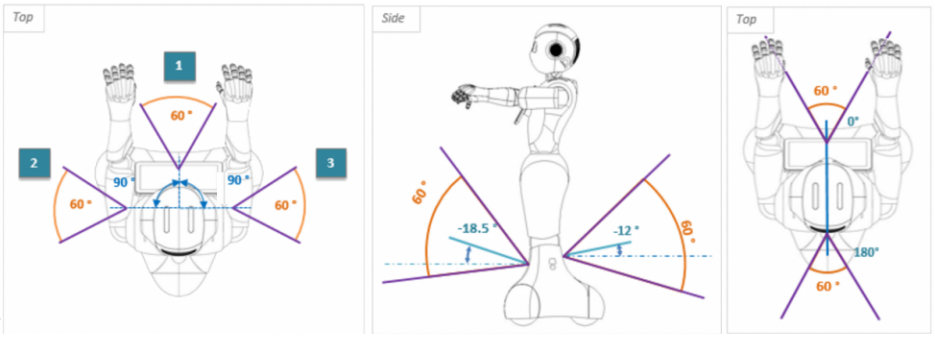

Figura 20- Campo di rilevamento del Pepper.

L'uscita del sonar è la distanza in metri dell'oggetto chiuso in prossimità del Pepper. Questo sensore non può fornire nessun'altra informazione.

Per quanto concerne i laser, il Pepper ha 3 set incorporati sotto la ginocchiera. Ogni laser è composto da 15 raggi, ma nel Wrapper ROS vengono aggiunti altri 16 raggi virtuali impostati a zero per le parti cieche del campo di rilevamento,

Il campo di rilevamento è impostato a 240 gradi di cui solo 180 gradi sono zone di rilevamento vero e proprio. La distanza massima di rilevamento è definita fino a 10 metri con un'altezza massima di 10cm per gli oggetti che i laser possono rilevare.

Ha inoltre due telecamere identiche situate nella parte superiore e inferiore della testa (fronte e bocca)

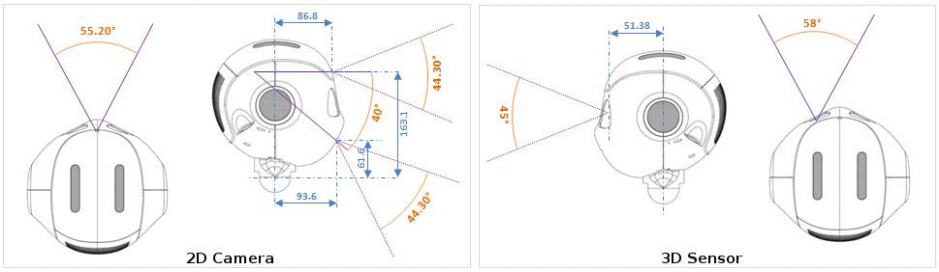

Figura 21- Campo visivo che entrambe le telecamere stereo producono.

Uno dei principali miglioramenti del design di Pepper rispetto ad altri modelli NAO prodotti da Aldebaran-Softbank Robotics è il sensore 3D posizionati negli occhi del robot.

Le telecamere di profondità è il modello ONE ASUS Xtion disponibile in commercio è che permette un campo di rilevamento come viene mostrato nella figura sovrastante.

4.4 Apprendimento/ Riconoscimento del volto

Il riconoscimento del volto è una tecnica di intelligenza artificiale, utilizzata in biometria per identificare o verificare l'identità di una persona a partire da una o più immagini che la ritraggono. Il robot Pepper invece utilizza il riconoscimento facciale per due eventi specifici nel comportamento guida:

1. Rilevare il partner per poi impegnarsi nel comportamento guida;
2. Rilevare il partner dopo che il rilevamento è stato interrotto;

Per fare ciò può essere utilizzata l'applicazione desktop multipiattaforma di Aldebaran Choreographe.

Choreographe è uno dei componenti del pacchetto software di Nao robot, in particolare robot Pepper. È un software di programmazione a blocchi, o

programmazione visuale. Questo significa che dispone di una interfaccia grafica intuitiva che lo rende particolarmente facile anche a programmatori non esperti.

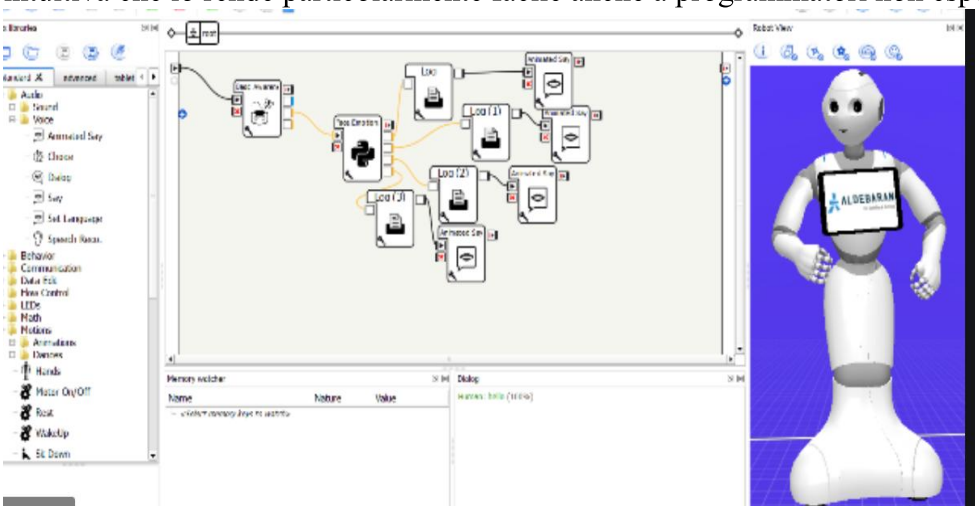

Figura 22- Pepper in Choreographe.

Per l'apprendimento/ riconoscimento del volto bisogna innanzitutto lanciare e caricare l'applicazione in Choreographe. Bisogna successivamente verificare che il robot Pepper sia impostato in modalità 'Head following mode'. A seguire mettere il partecipante in piedi davanti al Pepper e confermare che un nuovo volto umano viene rilevato. Eseguire dopo l'applicazione in Choreographe e attendere la fine del processo. Se dopo questa operazione si avranno led verdi sta a significare esito positivo altrimenti si avranno led rossi e indicano che Pepper non ha potuto imparare la faccia rilevata. In questo caso sono state testate due opzioni per migliorare il risultato:

Cambiare posizione;

Aumentare intensità della luce;

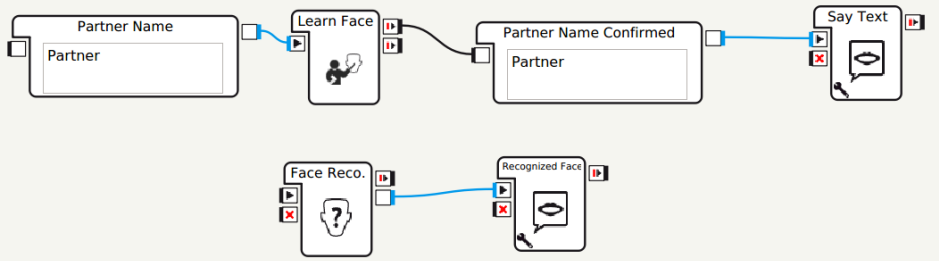

Figura 23-Schema del programma di apprendimento e riconoscimento del viso in Choreographe.

4.3 Pepper su Rviz

La prima applicazione che ho preso in considerazione per simulare i movimenti del robot è Rviz. Software di ROS dedicato alla visualizzazione di modelli tridimensionali, utilizzato per rappresentare i robot e il loro moto e rilasciato, come l'intera libreria a cui appartiene, sotto licenza BSD.

Rviz è uno sviluppatore e il suo compito principale è renderizzare oggetti a schermo. Una nota molto importante è il fatto riguardante gli aspetti della fisica del robot che non vengono presi in considerazione da questo programma.

Per visualizzare Pepper su Rviz, è sufficiente aprire un nuovo Rviz.

Digitare sul terminale di Ubuntu il seguente comando:

rosrun rviz rviz

e si avrà una situazione simile alla figura sottostante.

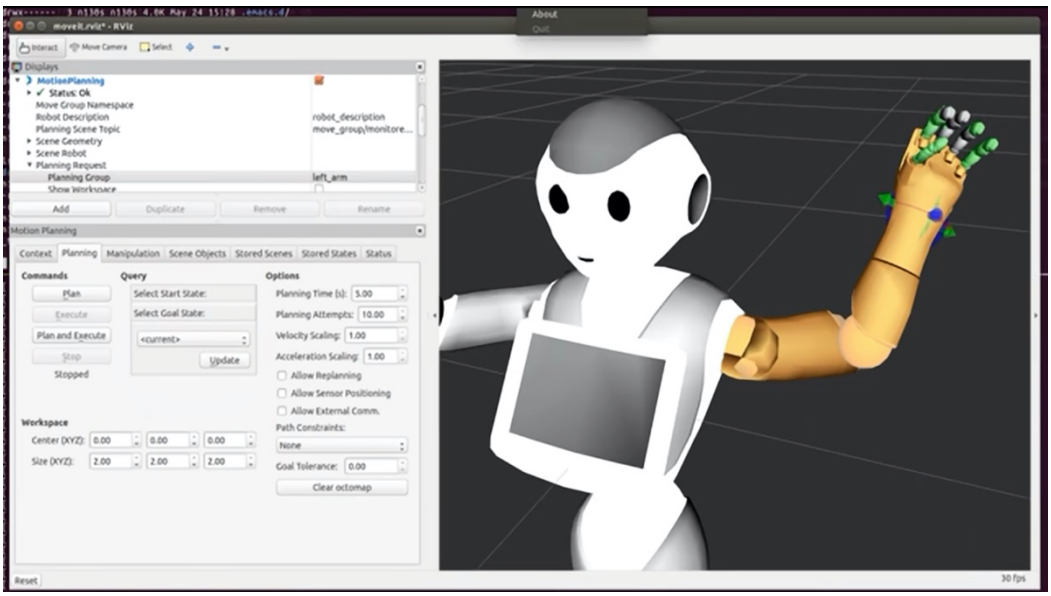

Figura 24- Esempio del Pepper in Rviz.

Una volta avviato Rviz, il programma mette in primo piano la visualizzazione del Pepper nel mondo tridimensionale e la finestra che si occupa del rendering.

Rviz presenta le seguenti sezioni:

- **Displays**: in questa finestra vengono elencati tutti gli oggetti che sono visualizzati nella schermata di rendering con la possibilità di rimuovere o inserire altri display(oggetti).
- **Tool Properties**: il pannello permette di associare una posizione di partenza e una posizione obiettivo ai due rispettivi topic di ROS.
- **Views**: in questa finestra è possibile scegliere il tipo di telecamera con cui visualizzare il mondo tridimensionale creato.
- **Selection**: pannello di fondamentale importanza per la descrizione dei link selezionati: per ciascuno di essi, vengono indicati l'orientamento e la posizione rispetto all'origine del sistema di riferimento fisso del mondo.

4.4 Pepper su Gazebo

La seconda applicazione presa in considerazione per questo lavoro è Gazebo. Gazebo è un simulatore 3D progettato per essere utilizzato con diversi tipi di robot, sia in ambienti interni che in ambienti esterni. Questo simulatore è stato progettato per lo sviluppo e conseguente testing di algoritmi per robot. La parte di simulazione è stata concepita per rappresentare fedelmente i movimenti dei robot reali, facilitando così la fase di testing presente durante lo sviluppo di un'applicazione robotica; grazie alla simulazione non è più necessario effettuare le prove sul robot reale, evitando così il sorgere di possibili problemi, quali gli errori hardware del robot o comportamenti inaspettati del software che rischiano di provocare danni al robot stesso. La simulazione è garantita dall'utilizzo di ODE (Open Dynamics Engine) come motore fisico e di OGREE(Object-Oriented Graphics Rendering Engine) per il rendering di tutti gli oggetti presenti nella scena tridimensionale. Inoltre si ha anche ODE, un motore fisico che si occupa principalmente della simulazione della dinamica dei corpi rigidi e della funzionalità riguardanti la presenza di collisioni fra gli oggetti inseriti nella scena.

Uno dei vantaggi di Gazebo deriva dalla sua architettura modulare: grazie ad essa, infatti, è possibile inserire nuove funzionalità attraverso lo sviluppo di plugin.

Per visualizzare Pepper su Gazebo bisogna digitare sul terminale di Ubuntu il seguente comando:

```
roslaunch pepper_gazebo_plugin pepper_gazebo_plugin_Y20.launch
```

Questo commando genererà Gazebo con Pepper su un campo robocup.

Inizialmente la simulazione sarà in mondalità pausa per consentire l'inizializzazione di tutti i controller.

Si avrà una situazione simile alla figura sottostante:

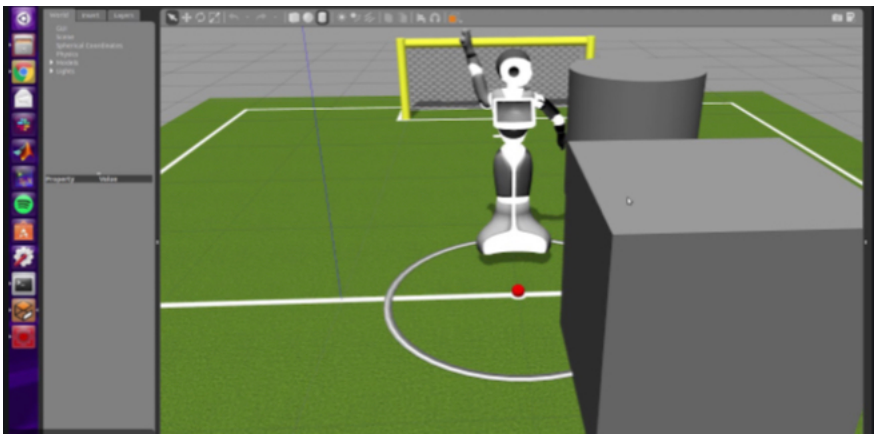

Figura 25- Pepper in Gazebo.

Per giunta, in Gazebo, l'ambiente viene caricato attraverso un file avente estensione .world, un file XML che contiene la descrizione dell'ambiente. Il file permette di personalizzare caratteristiche base del simulatore come i parametri fisici, la gravità, il materiale di cui dovrà costituirsi il suolo, la luce presente nell'ambiente e così via. Importante è il parametro /use_sim_time; esso porta ROS ad usare il tempo di simulazione pubblicato da Gazebo all'interno del topic/ clock, invece che andare ad usare l'orologio del sistema.

Quando ad esempio il Pepper sarà in esecuzione, infatti, le librerie ROS useranno l'orologio di sistema, quello riportato dal computer. Allo stesso tempo però quando si sta eseguendo la simulazione è sempre meglio avere un tempo stimato in modo che si possa avere controllo sulle accellerazioni, rallentamenti o gli stop.

Ultima nota importante, affinchè il robot possa venir simulato in Gazebo è indispensabile estendere il file URDF precedentemente creato andando ad aggiungere la mappatura dello stesso all'interno del simulatore.

4.5 Risultati preliminari sviluppati in ROS per Pepper

In quest'ultima parte verrà proposto un esempio applicativo di utilizzo del ROS per la movimentazione del robot caso di studio, Pepper, facendo riferimento a tutti gli argomenti illustrati nei capitoli precedenti. Il primo passo che ho effettuato è stato quello di andare sul sito di Aldebaran, perché il robot di studio è gestito proprio dal framework NAOqi. Fortunatamente un pacchetto ROS è fornito da Aldebaran per collegare parti specifiche dell'API di NAOqi come componenti di un sistema ROS. Il componente principale di questo pacchetto è incorporato all'interno del driver Naoqi che potrebbe quindi essere utilizzato anche per altri robot Softbank quali Nao o Romeo. Inoltre se prendiamo gli stati del giunto, la posizione del robot stesso e così via, essi sono pubblicati secondo gli standard ROS. L'eseguibile NAOqi viene fornito con un elenco di moduli principali e un'API pubblica con le sue funzionalità divise in gruppi come riporterò di seguito:

- **NAOqi Core** : moduli che gestiscono le comunicazioni, la gestione dei moduli, l'accesso alla memoria e il riconoscimento delle emozioni;
- **NAOqi Motion** : moduli che implementano animazioni, attività di navigazione e controllo a basso livello della posizione dei giunti e della velocità di base;
- **NAOqi Audio** : moduli che controllano il parlato animato, il riconoscimento vocale e la registrazione audio;
- **NAOqi Vision** : moduli che eseguono il rilevamento dei blob, l'acquisizione di foto e la localizzazione di base e il rilevamento dei punti di riferimento;
- **NAOqi People Perception** : moduli focalizzati sull'interazione uomo-robot, con rilevamento del volto, analisi dello sguardo e tracciamento delle persone;
- **Sensori NAOqi** : moduli per la lettura dei sensori laser, sonar e tattili

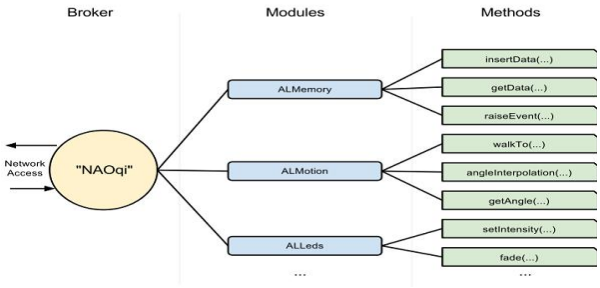

Figura 26-Schema Naoqi.

Oltre al driver di Naoqi, sono necessari diversi strumenti per andare a costruire un sistema completo del robot Pepper su ROS. I pacchetti usati sono stati:

- **I file di avvio**: include tutti i file di configurazione e di lancio che possono essere usati per avviare tutti i nodi necessari collegati al drivere NAOqi;
- **Modello del robot preso in considerazione**: questo pacchetto fornisce attraverso la descrizione e il file URDF la simulazione e la visualizzazione di Pepper all'interno di ROS;
- **Sensore Pepper specifico**: perché il driver NAOqi ha stabilito un collegamento tra il framework NAOqi e il ROS per tutti i robot che fanno parte della 'famiglia' Aldebaran/Softbank Robotics, sono necessari quindi dei file aggiuntivi per controllare i sensori e gli attuatori specifici dei singoli robot.

L'interfaccia ROS si registra come modulo nel NAOqi, quindi effettua chiamate con l'API NAOqi per leggere i sensori da Pepper e inviare comandi di velocità alla sua base.

Un punto importante per costruire un sistema di navigazione autonoma è quello di verificare l'odometria che viene fornita dal robot, nel mio caso le misurazioni odometriche sono pubblicate come argomento all'interno del pacchetto ROS.

Nelle immagini che seguiranno riporterò i primi risultati che ho ottenuto andando a muovere parti del corpo del Pepper su Gazebo rimanendo nell'ambiente iniziale preinstallato come nella figura del Paragrafo **4.4.**

Inizialmente il robot si presentava nella seguente posizione.

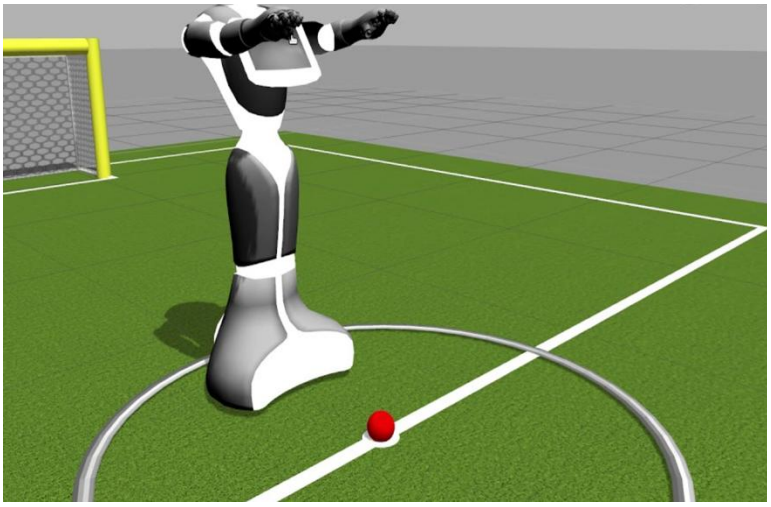

Figura 27-Posizione iniziale del robot Pepper.

Successivamente andando a muovere l'arto sinistro.

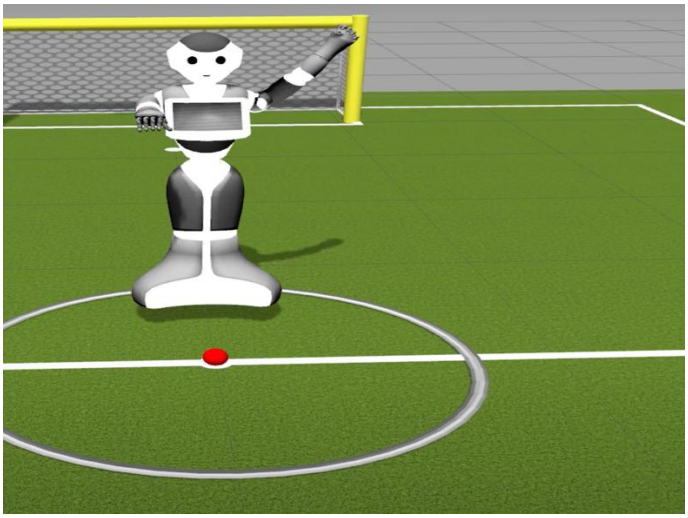

Figura 28-Prima movimento del robot Pepper.

Ho ottenuto questo risultato andando a dare comandi dal terminale del sistema operativo Linux, il terminale per ottenere il movimento è il seguente:

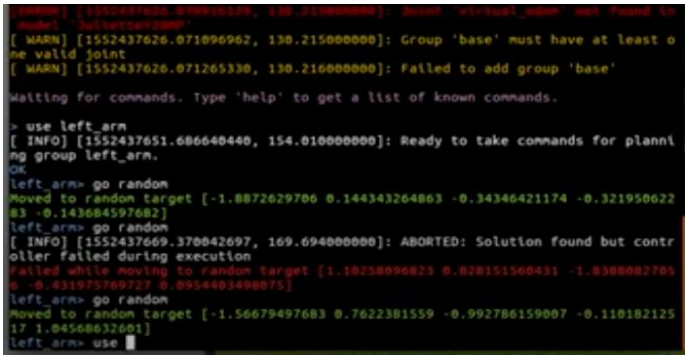

Figura 29-Terminale Ubuntu.

Dove si ha giustamente **left arm**, comando che serve per selezionare il braccio in cui dovrà avvenire il movimento. Con il comando **Go random** il robot muoverà il braccio in modo casuale. Lo stesso procedimento nel caso in cui volessimo muovere il braccio opposto. Nel caso in cui a muoversi saranno entrambi gli arti uso il comando:

both_arms

Altrimenti è possibile far muovere anche la testa, come rappresentato nella figura sottostante:

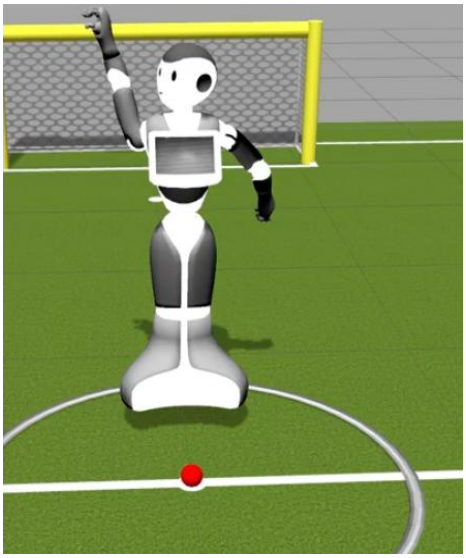

Figura 30- Movimento della testa su robot Pepper.

Andando anche questa volta sul terminale e digitare il comando:

use head

In questo modo si andrà a selezionare la parte del corpo relativa alla testa per permettere il movimento che avrà come nel caso del braccio il comando:

go random

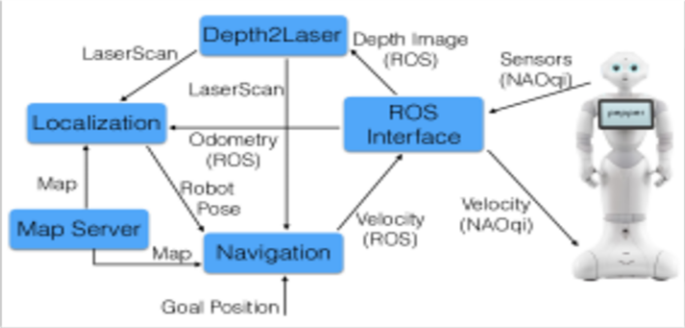

Figura 31- Schema navigazione autonoma Pepper.

Dopo aver letto i dati del sensore da Pepper, l'interfaccia ROS pubblica l'immagine di profondità e l'odometria utilizzando i tipi ROS standard. Un altro nodo sottoscrive l'immagine di profondità e la converte in una scansione laser. La comunità ROS fornisce un semplice pacchetto, **depthimage_to_laser scan** , che può eseguire questa trasformazione. Tuttavia, questo pacchetto è adatto per immagini acquisite da sensori 3D fissati e montati orizzontalmente, ad un'altezza ridotta, consentendo loro di rilevare facilmente gli ostacoli nel terreno. Pepper ha il suo sensore 3D nella testa, che può muoversi e cambiare orientamento. Pertanto, ho utilizzato altri due pacchetti, il primo che converte da immagini di profondità a nuvole di punti (**depth_image_proc**), e il secondo creando la scansione laser 2D dalla nuvola di punti 3D (**pointcloud_to_laserscan**). L'ultimo nodo può convertire i dati 3D dal suo frame di riferimento ad altri frame target, essendo flessibile sull'altezza minima e

massima dei punti da considerare per la conversione. Ci consente anche di eseguire la conversione quando la testa del robot è inclinata verso il basso.

Per la creazione della mappa, ho eseguito i seguenti passi. In primo luogo, viene avviato il nodo master ROS.

In secondo luogo, viene avviato il ROS su Pepper in modo che pubblichi i suoi dati. Successivamente si avvia l'immagine di profondità per puntare la nuvola di punti alla scansione laser. Dopo che tutti questi sono stati avviati completamente, ho eseguito l'algoritmo SLAM riproducendo il rosbag o il rosbag di una scena e successivamente viene salvata una mappa.

Fortunatamente gli stack di navigazione ROS includono il modulo gmapping per andare a creare una mappa.

Una volta creata una mappa, bisogna localizzare il robot all'interno; con l'uso di AMCL, avviando lo stimatore con il seguente comando:

```
$ rosrun amcl amcl scan: = / base_scan
```

Successivamente si apre RVIZ, impostando il frame fisso su /map e abilitando quindi la visualizzazione della scansione laser per andare a vedere la stima della posa iniziale. Per la navigazione all'interno della mappa creata bisogna innanzitutto aggiungere AMCL al file di avvio:

```
<node  name = "amcl"  pkg = "amcl"  type = "amcl" >
    <remap  from = "/ scan"  to = "/ base_scan"  />
</node>
```

Aggiungere un controller di movimento al robot

```
motion = MotionXYW()
```

```
motion.properties(ControlType = 'Position')

robot.append(motion)

motion.add_interface('ros', topic='/cmd_vel')
```

Infine per la navigazione utilizzerò il modulo ROS **move_base**.

La sottodirectory **morse_move_base** contiene valori standard per i parametri.

L'ho copiato nel mio nodo ROS e ho aggiunto il file di navigazione **nav.launch:**

```
<node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen" clear_params="true">



    <param name="footprint_padding" value="0.01" />

    <param name="controller_frequency" value="10.0" />

    <param name="controller_patience" value="100.0" />

    <param name="planner_frequency" value="2.0" />



    <rosparam file="$(find
morse_2dnav)/morse_move_base/costmap_common_params.yaml" command="load"
ns="global_costmap" />

    <rosparam file="$(find
morse_2dnav)/morse_move_base/costmap_common_params.yaml" command="load"
ns="local_costmap" />

    <rosparam file="$(find
morse_2dnav)/morse_move_base/local_costmap_params.yaml" command="load" />

    <rosparam file="$(find
morse_2dnav)/morse_move_base/global_costmap_params.yaml" command="load" />

    <param name="base_local_planner" value="dwa_local_planner/DWAPlannerROS"
/>

    <rosparam file="$(find
morse_2dnav)/morse_move_base/dwa_planner_ros.yaml" command="load" />

</node>
```

E infine ho eseguito il file di avvio:

```
roslaunch morse_2dnav nav.launch
```

5 Conclusioni

Lo scopo della tesi è quello di presentare ed esplorare ROS come framework per lo sviluppo di applicazioni di navigazione ed esplorazione autonoma di robot, fornendo anche un semplice esempio di applicazione su un robot umanoide, ovverosia il Pepper.

In questi mesi di tirocinio devo ammettere che ho avuto molte difficoltà con ROS, poiché risulta un campo molto complesso così come la robotica stessa non essendo un percorso che ho affrontato in modo diretto in questi anni di studio. Tuttavia è ormai diventato uno standard all'interno della comunità di sviluppo di applicazioni robotiche e negli anni avrà di sicuro un vasto sviluppo, perché i robot stessi avranno un largo impiego nella nostra quotidianità. Anche per questo motivo ho scelto di prendere con 'mano' questo lavoro e di capire meglio, anche se complicato, tutto quello che c'è dietro una 'semplice' macchina che imita l'aspetto e i movimenti dell'uomo, eseguendo operazioni in maniera autonoma e automatica. ROS ha sicuramente molti aspetti che lo rendono molto utilizzato rispetto ad altri framework quali OROCOS o Microsoft Robotics Studio perché essendo stato rilasciato sotto licenza BSD, quindi totalmente free e open-source, ha favorito certamente la comunità di ricercatori. Oggi ROS conta 13 versioni e questo ci fa recepire che è in continuo sviluppo e di conseguenza sono sempre maggiori il numero di piattaforme robotiche a supportarlo. Nel complesso però ha anche due principali limitazioni che ho riscontrato anche io, ovvero il fatto di usarlo in sistemi di piccole dimensioni, dove la disponibilità di

memoria è strettamente limitata e allo stesso tempo il fatto che funzionasse solo su sistemi Linux e avendo un pc IOS ho dovuto fare un dual Boot partizionando quindi in due l'hard disk. Per il futuro, la vera sfida di questo framework sarà appunto quello di poter sviluppare una versione che sia compatibile con tutti i sistemi operativi. Per quanto riguarda gli algoritmi di navigazione autonoma, la sfida è da una parte tecnologica, ovverosia la possibilità di avere sensori sempre più accurati e performanti, e dall'altra metodologica, ovverosia sviluppare algoritmi che siano computazionalmente sempre più efficaci ed efficienti per una loro applicazione più immediata ed auspicabilmente real-time.

Ringraziamenti

Eccomi giunto alla fine di questa tesi e di questi tre splendidi anni di università. Ho affrontato diverse prove, più o meno piacevoli, ma sono sempre riuscito a portare avanti il mio obiettivo con determinazione e passione.

Vorrei dedicare queste ultime pagine per ringraziare tutte le persone che in me hanno sempre creduto e che mi hanno sostenuto sia ne momenti di difficoltà sia in quelli felici e spensierati. Vorrei che questi ringraziamenti siano un punto di arrivo da una parte, ma anche un punto d'inizio, per il nuovo percorso che andrò ad affrontare nei prossimi due (spero) anni di carriera universitaria al fine di poter raggiungere nuovi traguardi importanti nella mia vita con tutti voi al mio fianco.

In primis vorrei ringraziare il prof Andrea Monteriù, relatore di questo mio lavoro, che ha dedicato il suo tempo ed impegno nella risoluzione di tutti i miei dubbi dandomi un aiuto anche a distanza e per il supporto costante durante questi mesi di tirocinio.

Un doveroso ringraziamento va ovviamente ai miei genitori, senza la quale non avrei neppure cominciato questa carriera; grazie per avermi sempre supportato e per non avermi fatto mancare mai nulla, vi devo tutto.

Un enorme grazie ad una persona unica e speciale, Daiana, la mia ragazza, che con amore, pazienza e fiducia mi ha sostenuto per questi anni con la speranza che continui a farlo a lungo.

Tutti i miei amici (fortunatamente sono tanti) che hanno avuto un peso (più o meno determinante) nel conseguimento di questo risultato.

Grazie NOT GANG siete stati il baricentro di questi tre anni, mitici e insostituibili coinquilini, per la compagnia, l'amicizia, le risate, le sfide a FIFA alla playstation dopo cena, per le pizze durante le partite di Champions League mentre si gufava la nostra nemica uve ma soprattutto per le accese discussioni ogni volt a che arrivava una bolletta del Gas.

Grazie a Simone, amico che ho conosciuto a Fermo tra i banchi dell'università ma con cui ho stretto un rapporto speciale. Abbiamo affrontato insieme questo cammino, passo dopo passo, giorno dopo giorno, superando tutte le difficoltà, festeggiando insieme ogni volta che si passava un esame. Grazie per essere stato sempre al mio fianco in ogni momento…ci aspettano altri due anni insieme lo zì

Grazie a tutti i miei amici di Fermo, colleghi di università e non. In particolare Davide (anche mio coinquilino), Tino, Giacomo, Angelica, Sara, Mariolina, Chiara, Peppino, Giorgio, Veronica, Moretti, Andrea, Christian, Federico, Fabrizio e spero di non aver dimenticato nessuno, vi ringrazio per tutto il tempo che mi avete dedicato.

Grazie agli amici del mio paese, Bomba; con alcuni di voi ho condiviso banchi di scuola con altri no, ma resterete sempre un pilastro della mia vita, amici di infanzia con cui sono cresciuto e con cui ho condiviso i miei giorni. Vi ringrazio per l'amicizia, per le risate, per le avventure e per le cavolate fatte insieme.

Grazie agli amici di Casoli, Valentina, Girolamo, Alessia, Giuseppe, essica, Giadina Lorenzo e Raffaele conosciuti recentemente ma con cui mi sono trovato bene fin da subito, vi ringrazio infinitamente per il vostro appoggio e per la vostra presenza.

Un ultimo ringraziamento, ma non per importanza, va a tutta la mia famiglia per esserci sempre stati.

Vi voglio bene.

‘’ Quanto manca alla vetta?’’ ‘’Tu sali e non pensarci’’ (F.W. Nietzsche)

Bibliografia

1 ROS Wiki, http://www.ros.org/it

2 Sito ufficiale, http://www.ros.org

3 ROS: an open-source Robot Operating System, http://cs.stanford.edu/people/ang/?portfolio=ros-an-open- source-robot-operating-system

4 Lentin oseph , onathan Cacace 'Mastering ROS for Robotics Programming' Second edition,2018 and first edition,2015.

5 A.S Group, ''Pepper'', 2017.

6 A.S Group, ''Aldebaran documentation: Technical overview,''2017.

7 Gazebo. http://www.gazebosim.org.

8 Rviz, https://github.com/ctanakul/pepper-simulation

9 A Gentle Introduction to ROS , ason M. O'Kane

10 SLAM, https://husarion.com/.