UNIVERSITA' POLITECNICA DELLE MARCHE
Facoltà di Ingegneria
Dipartimento di ingegneria dell'Informazione
Corso di Laurea Triennale in Ingegneria Informatica e dell'Automazione

---

TESI DI LAUREA

Studio e implementazione di algoritmi di localizzazione e navigazione autonoma per lo sviluppo di servizi assistivi attraverso personal robot

Study and implementation of autonomous localization and navigation algorithms for the development of assistive services through personal robots

Relatore:
Ing. A. Monteriù

Laureando:
Enrico Maria Sardellini

Correlatore:
Ing. L. Cavanini

A.A 2022/23
Dicembre 2023

RINGRAZIAMENTI

"Caro Enrico Maria, nonno Dino ti dice
più scuola e meno cartoni. Buon compleanno"

Sono le ultime parole che mio nonno mi scrisse in un bigliettino e che da allora le conservo in una busta che porto con me ad ogni esame. Sono cambiate molte cose da allora: gli interessi, le amicizie le idee, ma continuo a considerarmi ancora lo stesso ragazzino di quella lettera. Arrivare a scrivere questa tesi, che per alcuni è un traguardo da festeggiare, la sento come un qualcosa di assolutamente normale, obbligata da conseguire. E devo necessariamente ringraziare alcune persone per avermi facilitato la scalata.
In primis i miei genitori, Luigino ed Arianna, che mi hanno sostenuto sempre non facendomi mai pesare un insuccesso. Ringrazio anche mia sorella Elena, per non avermi mai disturbato mentre studiavo a casa nei primi mesi sotto covid, e nonna Ave per avermi sempre fatto trovare pronti pranzi e cene.
Passando agli amici dell'università ci tengo a ringraziare in particolar modo Federico S., Emanuele, Nicola, Pierandrea, Omar, Michelangelo, Andrea P., Eze, Marco, Alessandro. Mi rendo conto che ce ne sono anche altri che non ho citato, persone che ho conosciuto, frequentato i corsi e svolto esami insieme. Ma dovendo scegliere i più importanti, quelli che hanno avuto un ruolo attivo nello studio e/o nel divertimento sono per forza questi, e non gli ringrazierò mai a sufficienza.
Passando ai professori, devo sicuramente ringraziare Andrea Monteriù che mi ha seguito per completare al meglio il presente lavoro e Luca Cavanini che mi ha affiancato e sostenuto nelle difficoltà quotidiane, derivate soprattutto dalla mia inesperienza di qualsiasi argomento trattato nel tirocinio.
Vorrei dedicare anche qualche riga a coloro che sebbene non mi hanno aiutato con gli esami, mi hanno aiutato a non uscire completamente di senno sopra i libri e le slide, ovvero Ginevra, Michele, Federico B., Kevin, Andrea M.
Se questo traguardo non lo festeggio è perché sono consapevole che quello appena concluso non era altro che una collina, e davanti a me giace ancora una montagna, ma almeno, dopo 8 anni, posso uscire a riveder le stelle.

INDICE

CAPITOLO 1 – INTRODUZIONE

Il presente lavoro di tesi descrive l'implementazione di un sistema di localizzazione applicato al codice di un robot in simulazione. Tale tesi è basata sul lavoro svolto durante il tirocinio in cui si è cercato di implementare il navigation stack, che fornisce funzionalità per la navigazione autonoma dei robot, prima nel robot e poi, a causa di un problema, in un ambiente di simulazione. Lo scopo di questa tesi è di essere una guida rapida per poter ricreare l'ambiente di sviluppo per il robot risolvendo molti errori che si potrebbero incontrare. Per questo lavoro è necessario comprendere cosa sia la robotica e, in particolare, la localizzazione robotica. [1].
La robotica è la disciplina dell'ingegneria che studia e sviluppa metodi per consentire ad un robot di eseguire compiti specifici riproducendo in modo automatico il lavoro umano.
La localizzazione robotica è una componente fondamentale della robotica e dell'automazione che riguarda la capacità di un robot di determinare e tenere traccia della sua posizione e orientamento all'interno di un ambiente. Per effettuare una mappatura dello spazio circostante, il robot necessita di una serie di parametri come la direzione, la distanza percorsa, la velocità, accelerazioni e tempo trascorso, ottenuti tramite dei sensori che possono essere:

- Sistemi di navigazione: GPS, odometria, sensori inerziali ed altri strumenti che permettono al robot di determinare la sua posizione.

- Sensori ambientali: telecamere, lidar, scanner e sonar vengono utilizzati per percepire l'ambiente circostante.

- Mappatura: in combinazione con la localizzazione, permette di creare una mappa dell'ambiente in cui il robot opera.

- Algoritmi di localizzazione: algoritmi che elaborano i dati provenienti dai sensori per determinare la posizione e l'orientamento del robot.

Tale tesi si articola nella seguente struttura:

- Nel secondo capitolo si presentano i programmi e i software.

- Nel terzo capitolo si illustra l'architettura del nostro robot, Ohmni Telepresence Robot.

- Nel quarto capitolo si espone il sistema sviluppato ed i vari passaggi necessari al completamento.

- Nel quinto capitolo si trarranno le conclusioni del lavoro svolto, le problematiche affrontate ed un piccolo accenno ai possibili sviluppi successivi.

- Nel sesto capitolo si visualizza il codice.

- Nel settimo capitolo si elencano le fonti e i siti necessari per la comprensione di tale tirocinio.

CAPITOLO 2 – SOFTWARE

Nel seguente paragrafo si illustrano le caratteristiche dei software e dei programmi necessarie per futuri sviluppi e simulazioni.

2.1 VIRTUAL BOX

Oracle VM VirtualBox, spesso abbreviato in VirtualBox, è un software di virtualizzazione gratuito e open-source che consente agli utenti di creare e gestire macchine virtuali su un computer host. Consente inoltre di poter gestire una vasta gamma di sistemi operativi al loro interno senza dover necessariamente possedere un hardware fisico. [8].
Per questo lavoro è necessario lavorare con un particolare linguaggio ROS Melodic, che spieghiamo più avanti, il quale richiede una versione apposita di Ubuntu da far girare nella VirtualBox, ovvero la versione 18.04.6.

2.2 ROS MELODIC

ROS (Robot Operating System) è un sistema meta-operativo open-source dedicato ai robot. Contiene tutti i servizi che ci si aspetta da un sistema operativo, incluse astrazioni hardware, controllo dei device a basso livello, implementazione delle più comuni funzionalità, messaggistica tra processi e gestione di pacchetti. Prevede inoltre tools e librerie per ottenere, costruire, scrivere ed eseguire codice attraverso molteplici piattaforme. [2].
Il "grafico" di esecuzione di ROS è una rete peer-to-peer di processi, potenzialmente distribuiti tra più macchine, che sono in comunicazione attraverso una infrastruttura di una rete di messaggi. ROS è un framework costituito da nodi singolarmente personalizzabili e debolmente accoppiati in esecuzione. Questi nodi possono essere raggruppati in packages e stacks, che sono facilmente condivisibili e pubblicabili, inoltre si ha a disposizione un ottimo repository di codici necessario alla risoluzione di molteplici problematiche. Altre caratteristiche importanti sono:

- le librerie scritte con chiarezza e facilmente interpretabili.
- l'implementazione è immediata in C++, Python.
- tramite rostest è facile effettuare dei test in ROS.
- attualmente gira esclusivamente su piattaforme Ubuntu e Mac OS X.

ROS ha tre livelli concettuali:

1) **ROS Filesystem Level**.
   In questo livello sono presenti specialmente le risorse del ROS che si incontrano su disco:
   - Packages: sono l'unità principale per l'organizzazione del ROS, infatti in essi sono contenuti i processi runtime di ROS (i nodi), le dipendenze da librerie, i set di dati, i file di configurazione e altro necessario alla configurazione del ROS.
   - Metapackages: Sono dei packages che servono a rappresentare un gruppo.

- Package Manifests: Il manifesto (package.xml) contiene tutti i metadati relativi al package come nome, versione, descrizione e altri dettagli.

- Repositories: Una collezione di packages che condivide un comune VCS system.

- Message (msg) types: Le descrizioni dei messaggi, definiscono le strutture di dati per i messaggi inviati dal ROS.

- Service (srv) types: Le descrizioni dei servizi definiscono i dati richiesti e ottenuti per i servizi in ROS.

2) ROS Computation Graph Level.

Il Computation Graph è la rete peer-to-peer dei processi del ROS che elaborano dati in maniera concorrente. Anche qui andremo ad elencare i concetti principali, dando loro una breve spiegazione:

- Nodes: I nodi sono processi che eseguono codice, infatti un robot di solito include diversi nodi. Per esempio un nodo per il controllo del laser, uno per il controllo dei motori, uno per la localizzazione, uno per la pianificazione della traiettoria, uno per un'interfaccia grafica e così via.

- Master: Il Master in ROS consiste in un supervisore per l'intero Computation Graph, senza il Master i nodi non sarebbero in grado di trovarsi e quindi di comunicare l'un l'altro.

- Parameter Server: Permette ai dati di essere salvati tramite delle chiavi in un centro comune di raccolta dati.

- Messages: I nodi comunicano tra di loro passandosi dei messaggi. Un messaggio è una semplice struttura dati comprendente tipi di dati. I più primitivi (integer, floating point, boolean…) sono supportati, così anche gli array sono visti come dati primitivi.

- Topics: I messaggi sono trasportati attraverso un sistema di trasporto comune chiamato Publish/Subscribe. Un nodo invia un messaggio pubblicandolo (publish) su di un topic, che è un nome utilizzato per identificare il messaggio. Un nodo, che sarà interessato ad un certo tipo di dato, si metterà in ascolto (subscribe) sul topic desiderato. Ci possono essere ovviamente più publisher e subscriber per ogni singolo topic, e ogni singolo nodo può pubblicare e mettersi in ascolto di più topic. In generale i publisher e i subscriber sono indipendenti, l'idea infatti è quella di disaccoppiare la produzione di informazioni dal loro consumo.

- Services: Il modello dei topic (publish/subscribe) è un modello molto efficiente ma non nel caso in cui ci deve essere una interazione tra mittente e destinatario, queste interazioni sono spesso richieste in sistemi distribuiti. In tali casi si usa il modello basato sui services. Si parte definendo due tipologie di messaggio: una per la domanda e una per la risposta. Un nodo che invia una domanda attende il tipo di messaggio corrispondente alla risposta prima di proseguire nelle sue azioni.

- Bags: I bags rappresentano il formato di salvataggio di un'esecuzione in ROS. Ovvero si salva l'intera esecuzione di tutti i nodi del ROS e si rieseguono simulando l'esecuzione in

real time dei dati simulando tutta la struttura creata dal ROS nel momento dell'esecuzione istante per istante.

3) **ROS Community Level.**
   Questo livello include tutte le risorse che permettono a gruppi diversi di cambiare software e di condividerlo con altri. Le risorse includono distribuzioni del ROS, codici gratuiti, ROS wiki e tutto quanto necessario per una corretta assistenza.

La versione di ROS che è stata usata è la Melodic abbinata alla versione 18.04.6 di Ubuntu (Bionic).

2.3 GAZEBO

Gazebo è un ambiente di simulazione open-source ampiamente utilizzato per la robotica e la ricerca in robotica. Gazebo è uno strumento che consente agli sviluppatori di simulare l'ambiente in cui un robot o un veicolo autonomo opererebbe nella realtà. Gazebo offre un ambiente di simulazione 3D da semplici spazi chiusi ad ambienti esterni complessi al fine di testare e sviluppare algoritmi di controllo, software di intelligenza artificiale e sistemi di guida autonoma. Gazebo tiene anche conto di molti aspetti della fisica del mondo reale consentendo ai modelli di interagire con l'ambiente in maniera realistica, permettendo agli sviluppatori di simulare sensori come telecamere, lidar, microfoni e attuatori. Infine può essere integrato con diversi framework di robotica tra cui ROS. [8].

2.4 RVIZ

RViz è uno strumento di visualizzazione 3D integrato in ROS, fondamentale per visualizzare dati generati o elaborati dai nodi ROS. RViz fornisce un'interfaccia grafica per visualizzare varie informazioni sullo stato del robot e dell'ambiente circostante. [8].
RViz consente:

- di visualizzare il modello 3D del robot, compresi i sensori e gli attuatori, in tempo reale.
- di visualizzare le nuvole di punti acquisite da sensori come i sensori lidar.
- di mostrare mappe generate dal sistema di mapping e localizzazione del robot.
- di visualizzare le trasformazioni coordinate utili per integrare dati provenienti da diversi frame di riferimento.
- di interagire con il robot specificando obiettivi di navigazione o modificando la posizione virtuale della telecamera.

2.5 DOCKER

Docker è una piattaforma di virtualizzazione leggera e containerizzazione che consente agli sviluppatori di condividere, distribuire e gestire facilmente le applicazioni in contenitori. Un container è un ambiente autonomo e isolato in cui un'applicazione e tutte le sue dipendenze sono confezionate insieme senza interferire con altri container o con l'host. I container di Docker sono altamente portabili difatti possono essere eseguiti su qualsiasi sistema che supporti Docker, indipendentemente dall'host. Oltre ai container, c'è un altro aspetto chiave: le Immagini. Queste sono fondamentalmente dei pacchetti che contengono: il codice binario di un'app, le dipendenze necessarie, i metadati e le istruzioni su come eseguire l'app. Quando parliamo di un'immagine, parliamo di una configurazione che determina come deve essere fatto il sistema che andrà eseguito. Ogni immagine specifica come deve essere fatto il servizio che rappresenta, mentre ogni container è esattamente un'istanza di quel

servizio. Un'immagine è da intendere come una pila di livelli incrementali detti "layer". L'insieme ordinato dei layer costituisce il singolo oggetto immagine. [5].
Tuttavia Docker non è stato utilizzato. Capire come funziona serve a lavorare direttamente col robot e non in simulazione. Infatti il sistema operativo di Ohmni Telepresence è Android, il quale non supporta ROS. Di conseguenza il robot ci mette a disposizione un Docker per poter scaricare un'immagine di Ubuntu 18.04 con il quale andare a sviluppare il robot. Purtroppo tale possibilità non è possibile in quanto il robot presentava un problema. Di conseguenza si lavora esclusivamente in simulazione senza passare per Docker.

CAPITOLO 3 – HARDWARE

3.1 OHMNILABS

Il robot è stato sviluppato e assemblato dall'azienda OhmniLabs™, Inc., un'azienda di robotica con sede nella Silicon Valley incentrata sullo sviluppo, produzione e fornitura di robot sia per privati che per le aziende. Fu fondata nel 2015 da esperti di robotica, Jared Go e Tingxi Tan, nonché dall'imprenditore Thuc Vu. Nell'aprile del 2017 avvenne il primo lancio sul mercato di Ohmni Telepresence Robot e ad un anno di distanza, nel giugno del 2018, avvenne il lancio di Ohmni Developer Edition. [6].

3.2 OHMNI TELEPRESENCE ROBOT

L'Ohmni Telepresence è un robot alto circa 142,2 cm e con un peso di circa 11,5 kg. Poggia su una base stabile a 3 ruote con motori brushless da 30 W che gli permettono di raggiungere una velocità massima di 3,22 km/h. È dotato di una base di ricarica con contatti magnetici che fornisce, una volta caricata completamente la batteria, un'autonomia di 5 ore. Al centro è posizionato un altoparlante parlante professionale da 15 W, mentre in alto troviamo 2 telecamere ad alta definizione ed uno schermo con cui interagire. [4].

Figura 1 Ohmni Telepresence Robot

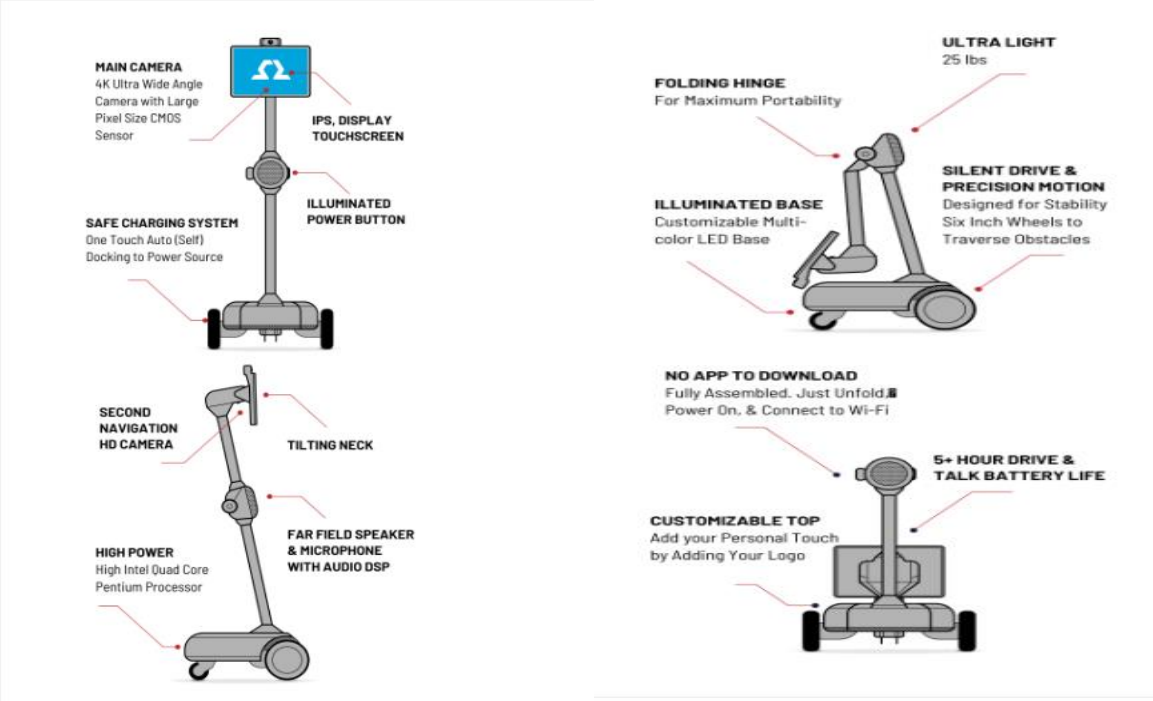

Figura 2 Descrizione componenti esterne del robot

3.2.1 ELETTRONICA DI BASE (BASE ELECTRONICS)

La scheda nella parte posteriore si chiama Core Board ed è responsabile della gestione, della ricarica, del bilanciamento, della potenza e della distribuzione dell'energia delle batterie a litio. Inoltre, pilota le luci LED alla base e fornisce un'uscita da 5 V 3 A che alimenta la scheda del processore principale. La scheda leggermente sotto è una scheda UP x86 integrata con il robot Ohmni. La Cpu principale è un processore quad core Intel Atom x5-Z8350 a 64 bit che tramite Turbo Burst funziona fino a 1,92 GHz. La scheda UP dispone di porte USB aggiuntive che possono essere utilizzate per collegare altre periferiche.

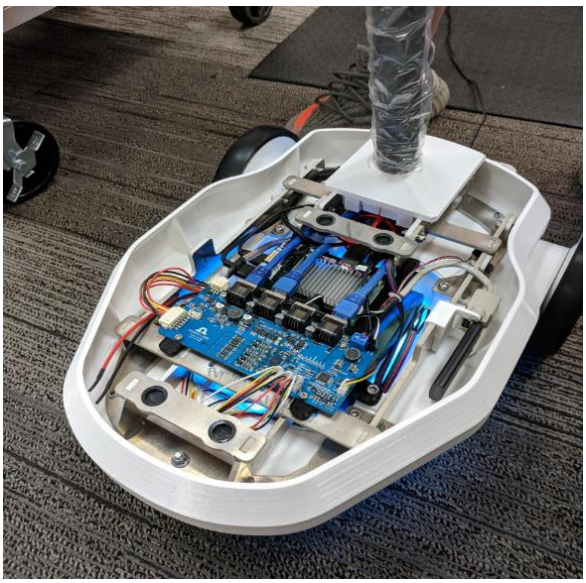

Figura 3 Elettronica alla base del robot

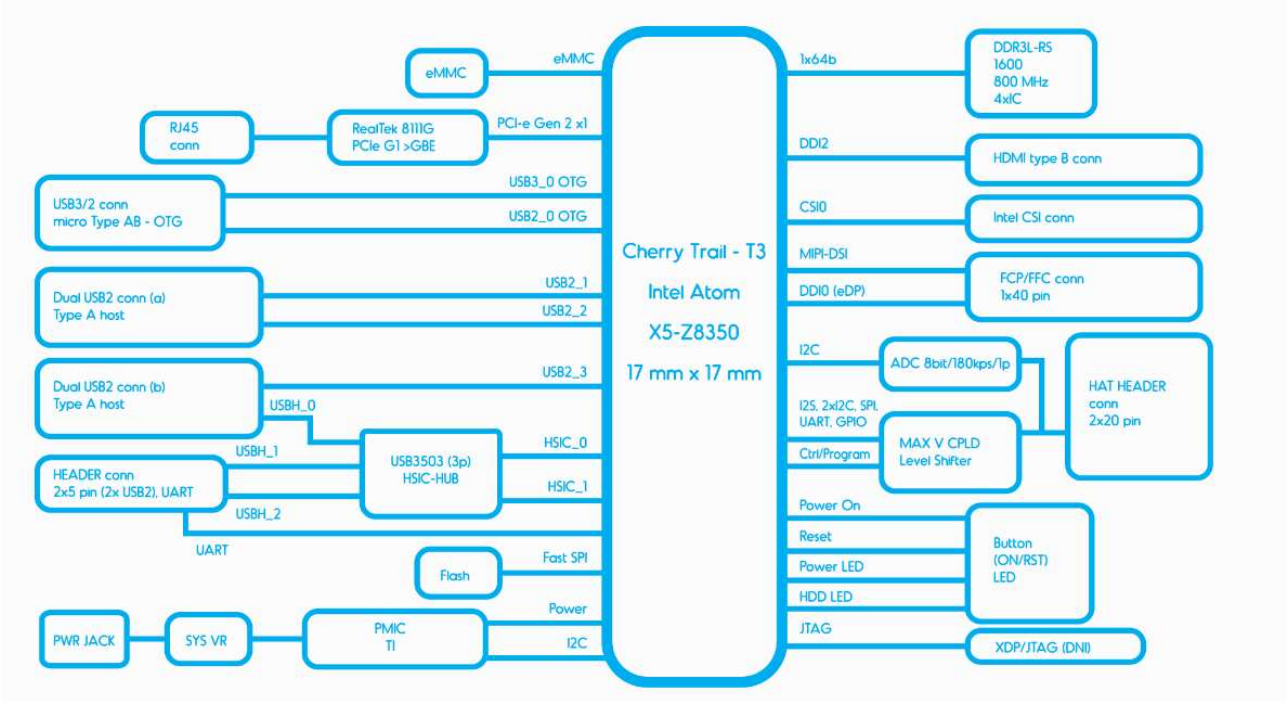

Figura 4 Driver Stack

3.2.2 UNITÀ (DRIVES)

I quattro jack Ethernet nella parte anteriore della Core Board non fungono effettivamente da porte Ethernet. Utilizzano gli 8 cavi Ethernet fisici e il fattore di forma come un bus schermato ad alta potenza con due pin per multidrop seriale. Due dei jack vanno alle ruote a destra e a sinistra. Il terzo fornisce energia alle parti superiori del robot. L'ultima è lasciata a disposizione per l'utilizzo in fase di sviluppo.

3.2.3 ELETTRONICA DEL COLLO/PARTE SUPERIORE

All'interno del collo sono presenti quattro porte USB che si collegano al touchscreen e alle fotocamere anteriori e inferiori. Possiede anche un convertitore ad alta potenza da 5V 3A che alimenta tutte queste periferiche. È presente anche un servo che controlla l'inclinazione dello schermo che è collegato direttamente al bus seriale multidrop ad alta potenza.

3.2.4 TOUCHSCREEN

Il touchscreen è realizzato su misura da OhmniLabs. È uno schermo da 10,1 pollici con una risoluzione da 1280 x 800 e un pannello touch con una scheda controller personalizzata. Ciò permette al touchscreen di comunicare con il resto del sistema USB e di controllare vari attributi del display. Richiede un'alimentazione USB e presenta un ingresso HDMI che supporta solo la risoluzione nativa ma in teoria è possibile collegare e visualizzare quasi tutti i dispositivi HDMI.

CAPITOLO 4 – SISTEMA SVILUPPATO

Lo scopo della qui presente tesi è quello di esporre tutti i passaggi al fine di replicare e poter ricominciare a sviluppare evitando problemi indesiderati o errori con il quale mi sono dovuto confrontare nel corso del tirocinio.
Verrà così suddiviso tale capitolo: inizializzazione dell'ambiente di sviluppo, sviluppo e modifica del codice del robot e del navigation stack con risoluzione dei relativi problemi, configurazione di RViz.

4.1 INSTALLAZIONI

Qui verranno mostrati tutti i passaggi da seguire nelle installazioni di una serie di software e codici tra cui: Docker, VirtualBox, ROS Melodic, Gazebo ed RViz (che sono stati spiegati nel capitolo 2). Non tutte le installazioni vanno eseguite, dipende con quale sistema operativo si opera.

4.1.1 CONNESSIONE AL ROBOT

Questo approccio è stato tentato nel tirocinio, ma a causa di un problema al robot non è stato possibile. Connettersi al robot permette di velocizzare di molto sia le installazioni richieste e quindi diminuisce la conoscenza richiesta dei programmi per poter lavorare, sia velocizza la fase di test non dovendoli fare in esecuzione ma andando direttamente a lavorare con il robot reale. [4].
I passaggi da seguire sono i seguenti:

1. **Abilitare funzione "sviluppatore".**
   Per accedere alla shell del robot, la prima operazione consiste nel recarsi nelle impostazioni del robot e cliccare sette volte sul bottone "Version name".
   Se cliccate 8 volte parte un countdown e la funzione sviluppatore si disattiva.
   Tale azione deve essere ripetuta ogni volta che si riaccende il robot.

2. **Abilitare "ADB".**
   Una volta abilitata la funzione sviluppatore, scorrerete in fondo alla pagina delle impostazioni e troverete degli elementi che prima non erano disponibili. Cercate una opzione "Enable ADB" e attivatela.
   Ciò permette di collegare il robot alla rete locale e di renderlo individuabile.

3. **Collegamento da Windows al robot.**
   Scaricare da tale link SDK Platform Tools
   https://developer.android.com/studio/releases/platform-tools.
   Ciò permetterà di abilitare il comando "adb" che permette ad un sistema operativo di interfacciarsi con un sistema Android.
   Una volta scaricato aprite il terminale e digitate "adb connect <robot_ip>".
   Sostituite <robot_ip> con il vero ip del vostro robot Ohmni che potete trovare sempre nel menu impostazioni del robot.
   Una volta che siete connessi, potete usare comandi come "adb shell", "adb push", "adb pull", che permettono rispettivamente di iniziare una shell, importare ed esportare file da e verso il robot.

4. **Ohmni Docker.**
   Adesso siete connessi al robot ma non potete fare molto.
   Infatti vi trovate in un sistema Android che non permette di sviluppare codice e necessitate di un sistema Ubuntu per poter andare avanti. [5].
   In nostro aiuto è stato sviluppato Ohmni Developer Edition che è un potente visualizzatore Docker che rende possibile avviare qualsiasi versione di Ubuntu all'interno di Ohmni.
   Per far ciò è necessario:

   - avviare una "adb shell" dal terminale di windows.
   - digitare "su" che permette di passare all'utente amministratore del sistema e avere i privilegi completi sul sistema.
   - digitare "dockerenv" che caricherà un'immagine dal repository di Docker Hub chiamata ohmnilabs/ohmnidev che è un'immagine di base di Ubuntu 18.04 con alcuni strumenti aggiuntivi installati.

   A questo punto, se tutto è andato bene, noterete che dal terminale si è passati da "ohmni_up:/ #" a "root@localhost:/home/ohmnidev#" che è una più tipica shell di Ubuntu. Quindi ora sarà possibile lavorare su Ubuntu in un sistema Android.

4.1.2 CREARE UNA MACCHINA VIRTUALE

Se, dopo aver eseguito il paragrafo precedente, compare "root@localhost:/home/ohmnidev#" allora non è necessario seguire questo paragrafo.
Se, invece, il vostro computer ha un sistema operativo Linux, è sufficiente cambiare la versione di Ubuntu e passare alla 18.04.6 e non è necessario seguire questo paragrafo.
Se invece appare un messaggio di errore che dice che non è disponibile Docker, e il vostro sistema operativo non è Linux, come nel caso del tirocinio, è necessario creare una macchina virtuale con una versione di Ubuntu specifica.
I passaggi da eseguire sono i seguenti:

1. **Scaricare e installare VirtualBox.**
   Accedendo al seguente link https://www.virtualbox.org/wiki/Downloads è possibile scaricare VirtualBox e successivamente installarlo. Per le spiegazioni su cosa sia VirtualBox, vi rimando al capitolo 2.1.

2. **Scaricare il codice per Ubuntu.**
   Accedendo al seguente link https://releases.ubuntu.com/18.04/ è possibile scaricare la versione di Ubuntu al fine di poter usare ROS Melodic per sviluppare il robot Ohmni.

3. **Creare la macchina virtuale.**
   A questo punto è possibile creare una macchina virtuale con il codice Ubuntu. Basta semplicemente:

   - cliccare in VirtualBox su "Nuova".
   - dargli un nome e assegnali in "immagine ISO" la versione di Ubuntu precedentemente scaricata.

- seguire i restanti passaggi in cui verranno richieste delle caratteristiche specifiche a vostra scelta come la password (è preferibile una password molto breve perché per ogni installazione che dovrà essere fatta, verrà richiesta), la memoria di base (meglio averla la più alta possibile in quanto alcune installazioni pesano) e i processori da affidargli (dipende da quanti il vostro computer ne possiede ma anche 2 bastano).

- una volta scelte le impostazioni, cliccate su "fine" e si creerà la macchina virtuale. Ora avete il vostro spazio Linux per sviluppare il robot.

4. **Problema col terminale.**
   Posso emergere dei problemi una volta all'interno dell'ambiente Linux.
   Il primo di questi è l'impossibilità di aprire il terminale. Per risolverlo:

   - Entrare nelle impostazioni.

   - Cliccare su "Language and region".

   - Cambiare il linguaggio con qualsiasi altro (possibilmente che si conosce).

   - Riavviare la macchina virtuale.

   Adesso la macchina virtuale si aprirà e potete rimettere la lingua precedente se volete.

5. **Problema amministratore.**
   Un'altra problematica è il fatto di non avere privilegi. Sebbene siate l'utente che ha creato e avviato la macchina virtuale, il sistema potrebbe non riconoscervi come amministratore e quindi determinati privilegi potreste non possederli. Al fine di risolvere ciò, ecco i passaggi da seguire:

   - Aprire un terminale .

   - Digitare "su": serve per cambiare dall'utente corrente al superutente o "root".

   - Digitare "apt install sudo": installa il pacchetto sudo nel sistema. Sudo è uno strumento che consente agli utenti autorizzati di eseguire comandi come un altro utente, di solito il superutente.

   - Digitare "usermod -aG sudo <user>": modifica i privilegi di <user>, che deve essere sostituito con il nome dell'utente che appare quando si avvia la macchina virtuale, permettendogli di eseguire i comandi sudo.

   - Digitare "exit": serve per passare stavolta dal superutente all'utente.

   - Digitare "reboot": riavvierà il sistema e salverà le modifiche fatte.

   Fatto ciò, si può procedere all'installazione del ROS Melodic che, senza privilegi, non era possibile.

4.1.3 INSTALLAZIONE ROS MELODIC

A questo punto, sia che vi trovate nel container Docker del robot sia sulla macchina virtuale, è necessario installare ROS Melodic, la versione del ROS compatibile con Ubuntu 18.04.6. [3].
Per maggiori dettagli su ROS Melodic, vi rimando al paragrafo 2.2 dove è ampiamente spiegato.
Ecco i passaggi necessari:

1. **Configura source.list.**
   Aprire un terminale e digitare il comando:
   "sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'"
   Questo comando configura il computer ad accettare software da package.ros.org

2. **Imposta le chiavi.**
   Le chiavi servono a verificare l'integrità e l'autenticità dei pacchetti ROS.
   Per far ciò è necessario inserire nel terminale i seguenti comandi:

   - Digitare "sudo apt install curl":
     permette di installare il programma curl che è uno strumento usato per trasferire dati da e/o verso un server utilizzando una serie di protocolli accettati come http, https, ftp, ecc.

   - Digitare "curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add –": questo comando scarica il file ros.asc che contiene le chiavi, le quali vengono aggiunte al sistema.

3. **Installazione di ROS.**
   L'installazione di ROS non è fissa, infatti esistono vari codici per l'installazione (totale, solo alcuni pacchetti, base, …).
   Per installare ROS è necessario:

   - Digitare "sudo apt update":
     è un comando per aggiornare i pacchetti Debian. In futuro, prima di qualsiasi installazione, è necessario aggiornare i pacchetti per evitare il sorgere di alcuni problemi.

   - Digitare "sudo apt install ros-melodic-desktop-full":
     è il comando per installare la versione completa del ROS provvista di ROS, rqt, RViz, librerie robotiche generiche simulatori 2D/3D e percezione 2D/3D.

   La procedura è abbastanza lunga e nel mentre potrebbe essere richiesto confermare alcune installazioni. È consigliato aspettare il la fine dell'installazione.

4. **Configurazione dell'ambiente.**
   È utile che le variabili dell'ambiente ROS vengano aggiunte automaticamente ogni qual volta venga aperto un terminale.

   - Digitare "nano ~/.bashrc" nel terminale il quale permette di accedere al bashrc.
   - Aggiungere alla fine del file "source /opt/ros/melodic/setup.bash".
   - Uscire e salvare. Basta premere CTRL+X, poi la Y e infine Invio per poter tornare nel terminale. Una volta qui, digitare "source ~/.bashrc" per eseguire direttamente nella shell corrente il file modificato.

5. **Dipendenze per la compilazione dei pacchetti.**
   Per creare e gestire il tuo spazio di lavoro ROS esistono veri strumenti e requisiti distribuiti separatamente. Uno di questi è Rosinstall usato spesso per scaricare molte directory sorgente di pacchetti di ROS con un solo comando. Ecco i passaggi da eseguire:

   - Installare Rosinstall:
     digitare "sudo apt install python-rosdep python-rosinstall python-rosinstall-generator python-wstool build-essential".
     Questo comando installa una serie di pacchetti necessari per la compilazione dei pacchetti in ROS.
   - Inizializzare Rosdep:
     digitare "sudo rosdep init" che lo inizializza e poi digitare "rosdep update" che aggiorna il database delle dipendenze.
     Rosdep ti consente di installare facilmente le dipendenze di sistema per le sorgenti che desideri compilare ed è necessario per eseguire alcuni componenti core in ROS.

6. **Creare un ambiente di lavoro per ROS.**
   Devono essere eseguiti i seguenti passi:

   - Creare il catkin_workspace:
     Per fare ciò è necessario digitare il comando "mkdir -p ~/catkin_ws/src" il quale crea una struttura di directory per lo spazio catkin.
   - Spostarsi nello spazio di lavoro:
     tramite "cd ~/catkin_ws/" è possibile spostarsi all'interno della cartella "catkin_ws".
   - Compilare i pacchetti ROS:
     per eseguire ciò e poter utilizzare i vari pacchetti di ROS è necessario sporsi nelle cartelle che contengono la cartella "src" ed eseguire il comando "catkin_make". Oltre a compilare e costruire i pacchetti, le sue altre funzioni sono: la gestione delle dipendenze tra i pacchetti, la creazione del "CMakeLists.txt" se viene effettuato per la prima volta in uno spazio di lavoro, la gestione delle varianti di Python.

- Verifiche finali:
  è necessario eseguire il comando “source devel/setup.bash” per preparare il tuo ambiente e infine verificare che la variabile dell'ambiente ROS_PACKAGE_PATH includa la directory in cui ti trovi tramite il comando “echo $ROS_PACKAGE_PATH”.

7. Librerie da scaricare.

È necessario inoltre scaricare una serie di librerie aggiuntive per il funzionamento del robot e del navigation stack. Queste serviranno ad evitare problemi legati o all’avvio dei file di lancio o nell’esecuzione di alcuni “catkin_make”. Prima di installare ogni nuova libreria è necessario eseguire il comando “sudo apt-get update”. [7].

- Slam_toolbox: fornisce una serie di algoritmi SLAM e strumenti che possono essere utilizzati per costruire mappe, stimare la posizione del robot e migliorare la navigazione autonoma. Il comando per scaricare tale libreria è
  “sudo apt-get install ros-melodic-slam-toolbox”.

- Libsdl-image1.2-dev: serve per caricare immagini in vari formati come superfici SDL. Il comando per scaricare tale libreria è “sudo apt-get install libsdl-image1.2-dev”.

- Tf2_sensor_msgs: contiene i messaggi di sensori per il sistema di trasformazione del tempo (TF2) in ROS. Il comando per scaricare tale libreria è
  “sudo apt-get install ros-melodic-tf2-sensor-msgs”.

- Move_base_msgs: contiene i messaggi di azione e i messaggi di stato utilizzati dal nodo move_base in ROS. Ad esempio, possono includere richieste per muovere il robot in una determinata posizione o per interrompere un'azione di navigazione.
  Il comando per scaricare tale libreria è
  “sudo apt-get install ros-melodic-move-base-msgs”.

- Driver_base: fornisce una base per l'implementazione di driver hardware per robot. Contiene classi di base e interfacce che possono essere utilizzate per sviluppare driver per diversi tipi di sensori e attuatori.
  Il comando per scaricare tale libreria è “sudo apt-get install ros-melodic-driver-base”.

- Navigation: contiene un insieme completo di strumenti e librerie per la navigazione autonoma di robot tra cui dei componenti chiave come move_base e amcl.
  Il comando per scaricare tale libreria è “sudo apt-get install ros-melodic-navigation”.

- Gmapping: fornisce un algoritmo di mappatura simultanea e localizzazione (SLAM) che consente al robot di creare una mappa dell’ambiente circostante.
  Il comando per scaricare tale libreria è
  “sudo apt-get install ros-melodic-slam-gmapping”.

4.1.4 INSTALLAZIONE GAZEBO

Gazebo è un ambiente di simulazione open-source utilizzato per la robotica e la ricerca in robotica. Per maggiori informazioni su cosa sia Gazebo, vi rimando al paragrafo 2.3 dove viene spiegato. Se si lavora fisicamente con il robot, tale installazione non è necessaria.
Per poterlo scaricare, nel caso in cui si voglia lavorare in simulazione, e poterlo far funzionare senza problemi è necessario eseguire i seguenti passaggi:

1. **Installazione.**
   Per procedere con l'installazione è necessario eseguire i seguenti comandi nel terminale:

   - Digitare "sudo apt update": che aggiorna l'elenco dei pacchetti nel sistema.

   - Digitare "sudo apt install gazebo" o "sudo apt install gazeb09-commons": in entrambi i casi viene scaricato e installato Gazebo ma il secondo codice permette di scaricare anche gli esempi e i modelli di Gazebo.

2. **Risoluzione problematiche.**
   Nel corso del tirocinio sono occorse alcune problematiche risolvibili modificando il bashrc. [7].

   - Digitare "nano ~/.bashrc" nel terminale il quale permette di accedere al bashrc.

   - Aggiungere alla fine del file "export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:/directory_personale/tb-simulation/ros_ws/src/tb_sim".

   - Aggiungere alla fine del file: "export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/directory_personale/tb-simulation/ros_ws/devel/lib".

   - In entrambe le aggiunte è necessario sostituire "directory_personale" con la posizione della vostra cartella "tb-simulation".

   - Uscire e salvare. Basta premere CTRL+X, poi la Y e infine Invio per poter tornare nel terminale. Una volta qui, digitare "source ~/.bashrc" per eseguire direttamente nella shell corrente il file modificato.

4.1.5 INSTALLAZIONE RVIZ

RViz è uno strumento di visualizzazione che fornisce un'interfaccia grafica per visualizzare varie informazioni sullo stato del robot e dell'ambiente circostante.
Per maggiori informazioni su cosa sia RViz, vi rimando al paragrafo 2.4 dove viene spiegato.
A seconda di quale pacchetto ROS Melodic è stato scaricato, RViz potrebbe già essere presente nella macchina virtuale. Non è un problema se viene scaricato 2 volte in quanto il sistema controllerà se è già presente ed in quel caso farà solo un aggiornamento dei pacchetti.
Per poterlo scaricare è necessario:

- Digitare "sudo apt update" per aggiornare i pacchetti nel sistema.

- Digitare "sudo apt-get install ros-melodic-rviz" per scaricarlo.

4.1.6 SCARICARE NAVIGATION STACK E CODICE DEL ROBOT

A questo punto rimane solo da aggiungere e scaricare nella macchina virtuale, sia di VirtualBox sia di Docker, il codice del navigation stack e il codice del robot per le simulazioni in Gazebo. Il navigation stack fornisce funzionalità per la navigazione autonoma dei robot all'interno di un ambiente, al fine di evitare ostacoli e di raggiungere destinazioni specifiche. Nel paragrafo 4.3 verrà fornita una spiegazione più approfondita.

Per poter cominciarci a lavorare è necessario seguire i seguenti passaggi:

1. **Installazione dei pacchetti.**
   Accedere alla seguente pagina di Github
   https://github.com/Ems01/tirocinio_ohmni_robot e scaricare sia il pacchetto del navigation stack sia il pacchetto con il codice del robot.

2. **Compilazione dei pacchetti.**
   Per iniziare a lavorare con i pacchetti è necessario eseguire i catkin_make.
   Affinché il sistema riesca a compilarli è necessario specificare nel bashrc la loro posizione.
   Quindi:

   - Digitare “nano ~/.bashrc” nel terminale il quale permette di accedere al bashrc.
   - Aggiungere alla fine del file “export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:/directory_personale/tb-simulation/ros_ws/src”.
   - Aggiungere alla fine del file: “export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/directory_personale/navigation/ws_navigation/workspace_carrozzina/src”.
   - In entrambe le aggiunte è necessario sostituire “directory_personale” con la posizione della vostra cartella “tb-simulation”.
   - Uscire e salvare. Basta premere CTRL+X, poi la Y e infine Invio per poter tornare nel terminale. Una volta qui, digitare “source ~/.bashrc” per eseguire direttamente nella shell corrente il file modificato.
   - Utilizzare il comando “cd” per passare alla directory delle cartelle che contengono la cartella “src”.
   - Digitare “catkin_make” per cominciare la compilazione.

Adesso è possibile lavorare con al simulazione del robot e avviare il navigation stack.

4.2 MODIFICHE AL CODICE DEL ROBOT

Al codice originale del robot e dei suoi file di lancio sono state modificate 2 parti in maniera tale da risolvere alcuni problemi che si verificano una volta che si comincia a lavorare con RViz.

4.2.1 EMPTY.LAUNCH

Questo file di lancio è progettato per configurare e avviare l'ambiente di simulazione Gazebo con il nostro robot all'interno. Le modifiche apportate al robot riguardano i due nodi presenti alla fine del file.

```
<!-- fake map frame for development
  <node pkg="tf2_ros" type="static_transform_publisher" name="fake_map"
    args="0.0 0.0  0.0 0 0 0 map odom" /> -->

<!---->
<node pkg="gmapping" type="slam_gmapping" name="slam_gmapping">
  <param name="base_frame" value="base_link"/>
  <param name="odom_frame" value="odom"/>
  <param name="map_frame" value="map"/>
  <param name="map_update_interval" value="1.0"/>
</node> -->
```

Figura 5 empty.launch

Inizialmente il file di lancio non presentava il nodo slam_gmapping, ma solo fake_map che utilizza il pacchetto "tf2_ros" per pubblicare una trasformazione statica tra due frame nel sistema di coordinate di ROS. Per qualche motivo, questo nodo generava un conflitto una volta avviato il navigation stack, che impediva al robot di spostarsi a causa delle due mappe differenti. Tuttavia, dopo averlo commentato, il mondo in Gazebo non si generava. Per risolvere questo problema si è introdotto slam_gmapping che gestisce il processo di mappatura SLAM, utilizzando i frame "base_link", "odom" e "map". Sono 2 i vantaggi nell'introduzione di slam_gmapping:

- Nessun problema con il navigation stack.
  Nel file di lancio "move_base.launch" del navigation stack è presente il medesimo nodo. In questo modo una volta che si avvia il navigation stack, si ferma il nodo slam_gmapping di empty.launch e si avvia quello di move_base.launch.

- Salvare la mappa.
  Se si esegue, in un altro terminale, il comando
  "rosrun map_server map_saver -f <mappa>" si produce una file pgm del mondo intorno al robot. L'utilità di ciò è di poter salvare tale mappa nel map_server del navigation stack e quindi di avere a disposizione sia la mappa dinamica di slam_gmapping, ma anche la mappa statica.

4.2.2 TB_SIM.URDF.XACRO

Il file tb_sim.urdf.xacro descrive la struttura del robot. In essi sono descritti i parametri e le proprietà del robot, i link principali del robot come “base_link” e “display_link”, i joint che collegano tra di loro i link, la definizione delle ruote e dei driver di controllo.
Si è aggiunto e commentato la seguente parte:

```
<!--
<link name="dummy"/>
<joint name="dummy_joint" type="fixed">
<parent link="dummy"/>
<child link="base_link"/>
</joint>-->


<link name="dummy"/>
<joint name="dummy_joint" type="fixed">
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <parent link="base_link"/>
    <child link="dummy"/>
</joint>
```

Figura 6 tb_sim.urdf.xacro

Il problema riguarda la relazione tra dummy e base_link. La sezione di codice sopra rappresenta il codice originale del robot, che generava il problema. Osservando le trasformazione (TF) in RViz e impostando base_link come “Fixed Frame”, apparivano dei warning a dummy, wheel_right_link e wheel_left_link che indicavano che non erano collegati con base_link. Tramite il comando “rosrun tf view_frames” è possibile visualizzare i frame di riferimento (coordinate frames) e le trasformazioni (TF) disponibili in un sistema ROS. Da qui si è notato che dummy e base_link non erano collegati. La modifica apportata a tb_sim.urdf.xacro fa sì che base_link sia il link padre rispetto al link figlio dummy. Con questa modifica, e con il commento del codice precedente, l’errore sparisce. Digitando “rosrun tf view_frames”, ne ho avuto la conferma. L'immagine seguente non può essere visualizzata in modo più definito a causa della sua elevata grandezza.

Figura 7 view_frames

4.3 NAVIGATION STACK

Il navigation stack è un insieme di software e algoritmi che collaborano per consentire al robot di pianificare il percorso, di localizzarsi e navigare in un ambiente complesso. [3]. Sebbene i navigation stack possano variare, essi presentano degli elementi in comune:

- Una mappa dell'ambiente, essenziale al fine di far muovere il robot.
- La localizzazione è un processo che determina la posizione del robot sulla mappa. Tale processo può dipendere da lidar, telecamere, odometria e altri strumenti.
- La pianificazione del percorso che decide come il robot si deve muovere dal punto di partenza al punto di arrivo stabilito.
- Un controllo del movimento, ovvero il sistema deve essere in grado di generare dei comandi per far muovere il robot in modo preciso.
- Algoritmi per gli ostacoli, ossia il robot deve poter essere in grado di evitare alcuni ostacoli e di aggiornare dinamicamente il percorso.
- Un'interfaccia utente per consentire agli sviluppatori di monitorare e interagire con il sistema.

Il navigation stack di questa tesi comprende una serie di nodi, e quindi di pacchetti ROS, che non devono essere scaricati, ma semplicemente compilati una volta scaricato il navigation stack da Github. Per avviare il navigation stack è necessario lanciare, con roslaunch, un file chiamato move_base.launch che avvia una serie di nodi aggiuntivi atti alla navigazione automatica del robot.

4.3.1 NODI

```
<launch>

  <!-- Run the map server -->
<node name="map_server" pkg="map_server" type="map_server" args="$(find mappe)/mappa_statica.pgm 0.05">
</node>

  <!-- Node for mapping
  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping">
        <param name="base_frame" value="base_link"/>
        <param name="odom_frame" value="odom"/>
        <param name="map_frame" value="map"/>
        <param name="map_update_interval" value="1.0"/>
  </node> -->

  <!--- Run AMCL -->
<include file="$(find amcl)/examples/amcl_diff.launch" />

  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
    <rosparam file="$(find cost)/costmap_common_params.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find cost)/costmap_common_params.yaml" command="load" ns="local_costmap" />
    <rosparam file="$(find cost)/local_costmap_params.yaml" command="load" />
    <rosparam file="$(find cost)/global_costmap_params.yaml" command="load" />
    <rosparam file="$(find cost)/base_local_planner_params.yaml" command="load" />
    <!-- <param name="odom_frame" value="tb_sim/odom"/>
         <param name="base_frame" value="base_link"/>
         <param name="cmd_vel_topic" value="cmd_vel"/>
         <param name="map_frame" value="map"/> -->
  </node>

</launch>
```

Figura 8 move_base.launch

L'avvio di move_base avvia una serie di nodi che svolgono precise funzioni e alcuni di questi sfruttano una serie di parametri che sono dei file ".yaml" di cui ne parliamo al paragrafo 4.3.2.

I nodi che vengono avviati sono:

- Map_server: fornisce servizi per caricare e pubblicare mappe. Nell'immagine in figura 8, viene caricata un'immagine chiamata "mappa_statica.pgm" che il sistema va a cercare nella cartella "mappe". Tale mappa è da considerarsi statica, ovvero non può cambiare e deve essere cambiata ogni volta qualora si cambi l'ambiente.

- Slam_gmapping: fornisce un algoritmo di mappatura simultanea e localizzazione (SLAM) che consente al robot di creare una mappa dell'ambiente circostante. Usa una serie di parametri quali base_link e odom per generare dinamicamente la mappa, a differenza di map_server che è statica.

- Amcl: anche detto "Adaptive Monte Carlo Localization" è un algoritmo di localizzazione per robot che utilizza il campionamento di particelle per stimare la posizione del robot in un ambiente. È particolarmente utile quando il robot si muove in ambienti sconosciuti o in evoluzione.

- Move_base: il suo scopo è quello di pianificare e controllare il movimento del robot da un punto di partenza a un punto di destinazione, evitando ostacoli lungo il percorso.

Un problema che potrebbe sorgere è legato all'uso simultaneo sia di map_server, sia di slam_gmapping. Infatti, nei casi in cui sono attivi entrambi, e la mappa di map_server è diversa da quella costruita in Gazebo o si cerca di cambiare la posizione iniziale del robot in RViz, il sistema impazzirà non riconoscendo più a quale mappa affidarsi. Il suggerimento è di commentare uno dei due nodi e riattivarlo a seconda delle necessità.

4.3.2 FILE YALM

Move_base avvia una serie di file yaml che devono essere modificati a seconda delle dimensioni e topic del nostro robot. Nell'ordine:

1. **Costmap_common_params.yaml (local_costmap & global_costmap).**
   Il navigation stack utilizza due costmap per memorizzare informazioni sugli ostacoli nel mondo. Una è usata per la pianificazione globale, quindi per la creazione di piani a lungo termine sull'intero ambiente, mentre l'altra è utilizzata per la pianificazione locale e l'evitamento degli ostacoli.

```
obstacle_range: 2.5
raytrace_range: 3
footprint: [[-0.1765, -0.2185], [-0.1765, 0.2185], [0.1765, 0.2185], [0.1765, -0.2185]]
#robot_radius: 1
inflation_radius: 0.55
observation_sources: laser_scan_sensor #point_cloud_sensor
laser_scan_sensor: {sensor_frame: base_link, data_type: LaserScan, topic: scan, marking: true, clearing: true}
#point_cloud_sensor: {sensor_frame: base_link, data_type: PointCloud, topic: scan, marking: true, clearing: true}
```

Figura 9 costmap_common_params.yaml

Nel dettaglio:

- Obstacle_range: determina la massima distanza della lettura del sensore che causerà l'inclusione di un ostacolo nella costmap.
- Raytrace_range: determina la distanza a cui verrà eseguito il raytrace dello spazio libero dato dalla lettura del sensore.
- Footprint: rappresenta la sagoma e la dimensione del robot. Nel caso di Ohmni Telepresence Robot, la sua base ha dimensione 35,3 x 43,7. Considerando il centro della base come coordinate (0,0), le posizioni degli angoli sono mostrate nel footprint.
- Robot_radius: può essere usato se il robot è circolare e ne indica il raggio.
- Inflation_radius: il robot considera tutti i percorsi i cui ostacoli rimangono a tale distanza o superiore.
- Observation_source: è una lista di sensori che passano informazioni alla costmap.
- Laser_scan_sensor: è definito da una serie di parametri tra cui "sensor_name" che è il nome del frame di coordinate del sensore (nel mio caso base_link); "data_type" che deve essere impostato su LaserScan o PointCloud a seconda del messaggio che utilizza il topic; "topic_name" deve essere il nome del topic su cui il sensore pubblica i dati (nel mio caso scan); "marking" e "clearing servono ad aggiungere o cancellare informazioni sugli ostacoli della costmap.

2. Global_costmap_params.yaml

Questo file contiene le opzioni di configurazione specifiche per la costmap globale.

```
global_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 5.0
  static_map: true
  width:  80
  height: 80
  resolution: 0.05
  rolling_window: false
```

Figura 10 global_costmap_params.yaml

Nel dettaglio:

- Global_frame: definisce in quale frame di coordinate la costmap deve funzionare.
- Robot_base_frame: definisce il frame di coordinate a cui la costmap dovrebbe far riferimento per la base del robot.
- Update_frequency: determina la frequenza, in Hz, con cui la costmap eseguirà il suo aggiornamento.

- Static_map: stabilisce se la costmap dovrebbe inizializzarsi basandosi o meno su una mappa fornita dal map_server. Se la mappa non è statica, è necessario mettere "false".

- Width, height, resolution: impostano la larghezza, l'altezza e la risoluzione della costmap.

- Rolling_window: se è "true", la costmap rimarrà centrata attorno al robot mentre il robot si sposta attraverso il mondo.

3. **Local_costmap_params.yaml**
Questo file contiene le opzioni di configurazione specifiche per la costmap locale.

```
local_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 10.0
  height: 10.0
  resolution: 0.05
```

Figura 11 local_costmap_params.yaml

Nel dettaglio:

- Global_frame: definisce in quale frame di coordinate la costmap deve funzionare.

- Robot_base_frame: definisce il frame di coordinate a cui la costmap dovrebbe far riferimento per la base del robot.

- Update_frequency: determina la frequenza, in Hz, con cui la costmap eseguirà il suo aggiornamento.

- Publish_frequency: determina la frequenza, in Hz, con cui la costmap pubblica informazioni di visualizzazione.

- Static_map: stabilisce se la costmap dovrebbe inizializzarsi basandosi o meno su una mappa fornita dal map_server. Se la mappa non è statica, è necessario mettere "false".

- Rolling_window: se è "true", la costmap rimarrà centrata attorno al robot mentre il robot si sposta attraverso il mondo.

- Width, height, resolution: impostano la larghezza, l'altezza e la risoluzione della costmap.

4. **Base_local_planner_params.yaml**
Il base_local_planner è responsabile del calcolo dei comandi di velocità da inviare alla base mobile del robot. È necessario impostare alcune specifiche il base alle specifiche effettive del nostro robot per far funzionare tutto.

```
TrajectoryPlannerROS:
  max_vel_x: 0.8
  min_vel_x: 0.1
  max_rotational_vel: 0.8
  min_in_place_vel_theta: 0.3

  escape_vel: -0.2
  sim_time: 2.0
  path_distance_bias: 0.6
  goal_distance_bias: 0.6


  acc_lim_th: 3.2
  acc_lim_x: 2.5
  acc_lim_y: 2.5

  holonomic_robot: true
  meter_scoring: true
```

Figura 12 base_local_planner_params.yaml

Nel dettaglio:

- Prima sezione: definisce i limiti di velocità del robot.
- Escape_vel: è la velocità di fuga in situazioni di escape.
- Sim_time: è il tempo di predizione per la simulazione del percorso.
- Path_distance_bias: è la ponderazione della distanza del percorso nella pianificazione del percorso.
- Goal_distance_bias: è la ponderazione della distanza dall'obiettivo nella pianificazione del percorso.
- Terza sezione: definisce i limiti di accelerazione del robot.
- Holonomic_robot: indica se il robot può muoversi indipendentemente lungo tutte le direzioni.

- Meter_scoring: indica che scala i pesi in base all'unità di misura in metri invece che in celle della griglia.

4.3.3 PROBLEMI E RISOLUZIONI

Il navigation stack con cui si lavora, è stato sviluppato dal collega Corredani durante il suo tirocinio, e modificato al fine di poterlo riusare in simulazione con un robot differente. Durante il tirocinio sono emersi una serie di problemi a riguardo.

1. **Non trova la cartella Mappe.**
   Per risolvere questo problema è necessario:

   - Digitare "nano ~/.bashrc" nel terminale il quale permette di accedere al bashrc.
   - Aggiungere alla fine del file "export ROS_PACKAGE_PATH=$ROS_PACKAGE_PATH:/directory_personale/navigation/ws_navigation/src".
   - È necessario sostituire "directory_personale" con la posizione della vostra cartella "navigation" oppure sostituire tutto ed indicare la cartella in cui sono salvate le mappe.
   - Uscire e salvare. Basta premere CTRL+X, poi la Y e infine Invio per poter tornare nel terminale. Una volta qui, digitare "source ~/.bashrc" per eseguire direttamente nella shell corrente il file modificato.

2. **Parametri deprecati.**
   Durante i primi lanci sono emersi di vari parametri deprecati. Questi erano "/map" nel move_base.launch, "/odom" in local_costmap_params.yaml e "/map" in global_costmap_params.yaml. Per risolvere tale errore è stato necessario eliminare la "/" dai vari topic per poter avviare il move_base

3. **Mappa vecchia.**
   Inizialmente veniva caricata una mappa vecchia di un corridoio e il robot la prendeva come mappa statica. Siccome però il robot avvia il nodo slam_gmapping per poter avere una mappa dinamica, il robot aveva due mappe differenti e non rispondeva. Ogni volta è necessario quindi:

   - Avviare il codice del robot che abilita il nodo slam_gmapping.
   - Attraverso il pulsante "build editor" di Gazebo è possibile creare un ambiente chiuso come una stanza o un corridoio.
   - Far muovere il robot. Siccome il robot presenta un angolo cieco e non può osservare un intero ambiente, soprattutto se molto grande, è possibile usare "rqt_robot_steering" per farlo muovere per un po' in giro ma controllato da voi.
   - A questo punto, se su RViz la mappa è stata tutta mappata, si può salvare e la mappa in un file pgm.
   - Spostare la mappa e il file ottenuti all'interno della cartella "Mappe" e sostituire il nome della mappa nel move_base.launch con il nome scelto.

- A questo punto è possibile avviare il move_base.launch, ma si genererà un problema in RViz.
- Per risolvere tale problema è necessario interrompere in nodo slam_gmapping ed è infine possibile lavorare in un ambiente statico generato da Gazebo.

Il codice per poter eseguire ciò è presente nell'appendice C.

4.4 IMPOSTAZIONI DI RVIZ

RViz è un potente strumento di visualizzazione che può essere utilizzato per molteplici scopi, ma per poterlo usare con il navigation stack è necessario prima configurarlo. Questo include: impostare la posizione del robot per un sistema di localizzazione come amcl, visualizzare tutte le informazioni di visualizzazione fornite dallo stack di navigazione e inviare obiettivi allo stack di navigazione tramite rviz. [3].

4.4.1 DISPLAY

Per la configurazione di RViz è necessario aggiungere una serie di display che abiliteranno uno specifico topic. Ciò deve essere fatto la prima volta, dopo aver lanciato il move_base.launch, altrimenti il sistema non troverà i topic che devono essere inseriti. Devono essere aggiunti i seguenti display:

1. **Static map.**
   Permette di visualizzare la mappa statica che viene servita dal map_server.

   Nome display: Static map
   Tipo display: Map
   Topic: /map

2. **Particle cloud.**
   Visualizza la nuvola di particelle utilizzata dal sistema di localizzazione del robot. La dispersione della nuvola rappresenta l'incertezza del sistema di localizzazione sulla posa del robot. Una nuvola molto dispersa riflette un'alta incertezza, mentre una nuvola condensata rappresenta una bassa incertezza.

   Nome display: Particle cloud
   Tipo display: Pose array
   Topic: /particlecloud

3. **Robot footprint.**
   Mostra a schermo un poligono che dipende dal footprint dichiarato nel costmap_common_params.yaml.

   Nome display: Robot footprint
   Tipo display: Polygon
   Topic: /move_base/local_costmap/footprint

4. **Obstacles.**
   Visualizza gli ostacoli che il navigation stack vede nella costmap. Per evitare collisioni, il poligono del robot non deve mai intersecare una cella che contiene un ostacolo.

   Nome display: Obstacles
   Tipo display: Grid cell
   Topic: /move_base/local_costmap/obstacles

5. **Inflated obstacles.**
   Visualizza gli ostacoli nella costmap del navigation stack gonfiati dal raggio inscritto del robot. Per evitare collisioni, il punto centrale del robot non dovrebbe mai sovrapporsi a una cella che contiene un ostacolo gonfiato.

   Nome display: Inflated obstacles
   Tipo display: Grid cell
   Topic: /move_base/local_costmap/inflated_obstacles

6. **Global plan.**
   Visualizza la porzione del piano globale che il piano locale sta attualmente seguendo.

   Nome display: Global plan
   Tipo display: Path
   Topic: /move_base/TrajectoryPlannerROS/global_plan

7. **Local plan.**
   Visualizza la traiettoria associata ai comandi di velocità attualmente impartiti alla base dal piano locale.

   Nome display: Local plan
   Tipo display: Path
   Topic: /move_base/TrajectoryPlannerROS/local_plan

8. **Planner plan.**
   Visualizza il piano completo per il robot, calcolato dal piano globale.

   Nome display: Planner plan
   Tipo display: Path
   Topic: /move_base/NavfnROS/plan

9. **Current goal.**
   Visualizza la posizione dell'obiettivo che il navigation stack sta cercando di raggiungere.

   Nome display: Current goal
   Tipo display: Pose
   Topic: /move_base/current_goal

4.4.2 POSE ESTIMATE & NAV GOAL

Dopo aver aggiunto i precedenti display, è necessario modificare i topic di "Tool properties" e di "Global options". In particolare:

- Il topic di "2D Pose Estimate" deve essere /initialpose
- Il topic di "2D Nav Goal" deve essere /move_base_simple/goal
- Il "Fixed Frame" deve essere impostato su "map"

Fatto ciò è necessario salvare tale configurazione per non dover ogni volta reinserire i display. All'avvio della simulazione del robot saranno già presenti, ma non utilizzabili fino all'avvio del navigation stack.
Ora sarà possibile far muovere il robot. Tramite il bottone "2D Pose Estimate" è possibile generare una freccia sulla mappa, la cui base è il punto di partenza del robot e il versore è la direzione verso cui punta il robot. Stessa cosa per "2D Nav Goal" che genererà una freccia la cui base sarà il punto di arrivo del robot e il versore sarà la direzione verso cui dovrà fermarsi il robot.

CAPITOLO 5 – CONCLUSIONI

Nel seguente capitolo verranno presentate tutte le conclusioni e verrà fatto il punto della situazione sull'intero progetto. Per prima cosa verrà presentato lo stato finale del progetto, mostrando i punti di forza e le problematiche, poi tutte le problematiche riscontrate durante il percorso e infine analizzare possibili evoluzioni e problematiche che dovranno essere risolte in futuro.

5.1 STATO DEL PROGETTO

Dopo il periodo di lavoro sul progetto si può affermare che gli sforzi sono stati ripagati quasi a pieno, che le esperienze fatte e le conoscenze apprese sono state molto utili e soddisfacenti, anche se di difficile comprensione almeno nelle fasi iniziali. Lo scopo del tirocinio era l'implementazione del navigation stack per poter far muovere un robot in maniera autonoma. La prima fase del lavoro è stata dedicata al collegamento del robot tramite il terminale del computer. Poi, per necessità, si è passati a lavorare con il codice del robot in simulazione. La fase successiva è stata dedicata a modificare il codice del navigation stack, e del sistema, per poterlo utilizzare insieme al codice del robot. Successivamente, a lancio riuscito del navigation stack, si è lavorato principalmente alla risoluzione di conflitti tra i due codice lanciati e alla configurazione in RViz. La parte finale del progetto è stata dedicata interamente al nodo "move_base". Infatti questo nodo è responsabile di tracciare il percorso da seguire con tutte le informazioni a sua disposizione e successivamente avviare il movimento per poter far muovere il robot. Nonostante il robot riceva sia la posizione iniziale da cui partire e ci si sposti, sia la posizione finale e crei il percorso da seguire, sia lo status del robot appaia ricevere l'obiettivo, il robot non si muove. La difficoltà è dovuta anche al fatto che, dal terminale di lancio di "move_base.launch", non appaiono né messaggi di errori né warning che possano aiutare a comprendere come risolvere il problema.

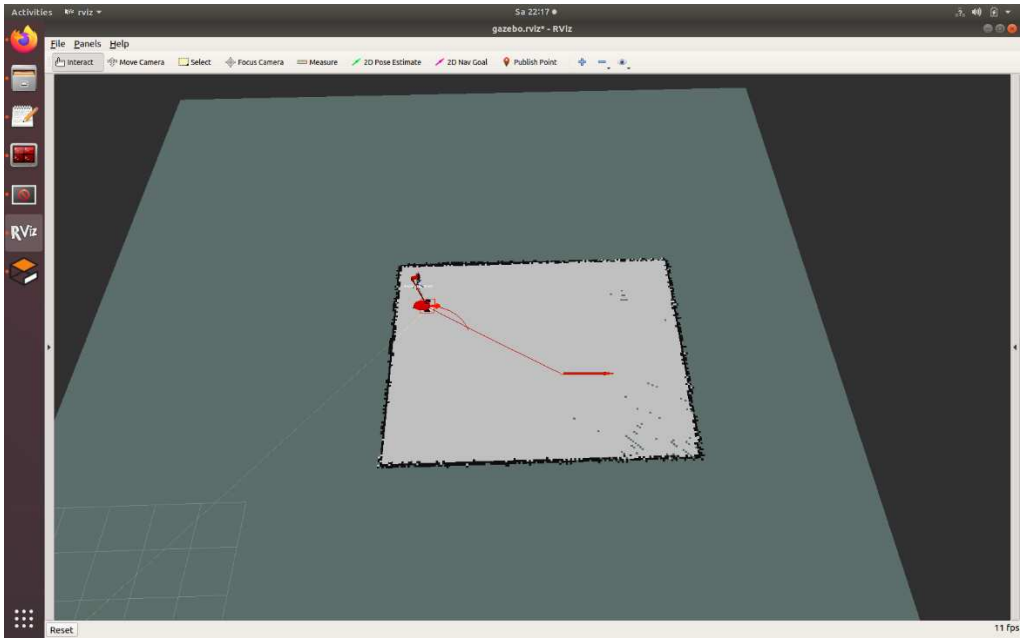

Figura 13 visualizzazione percorso in RViz

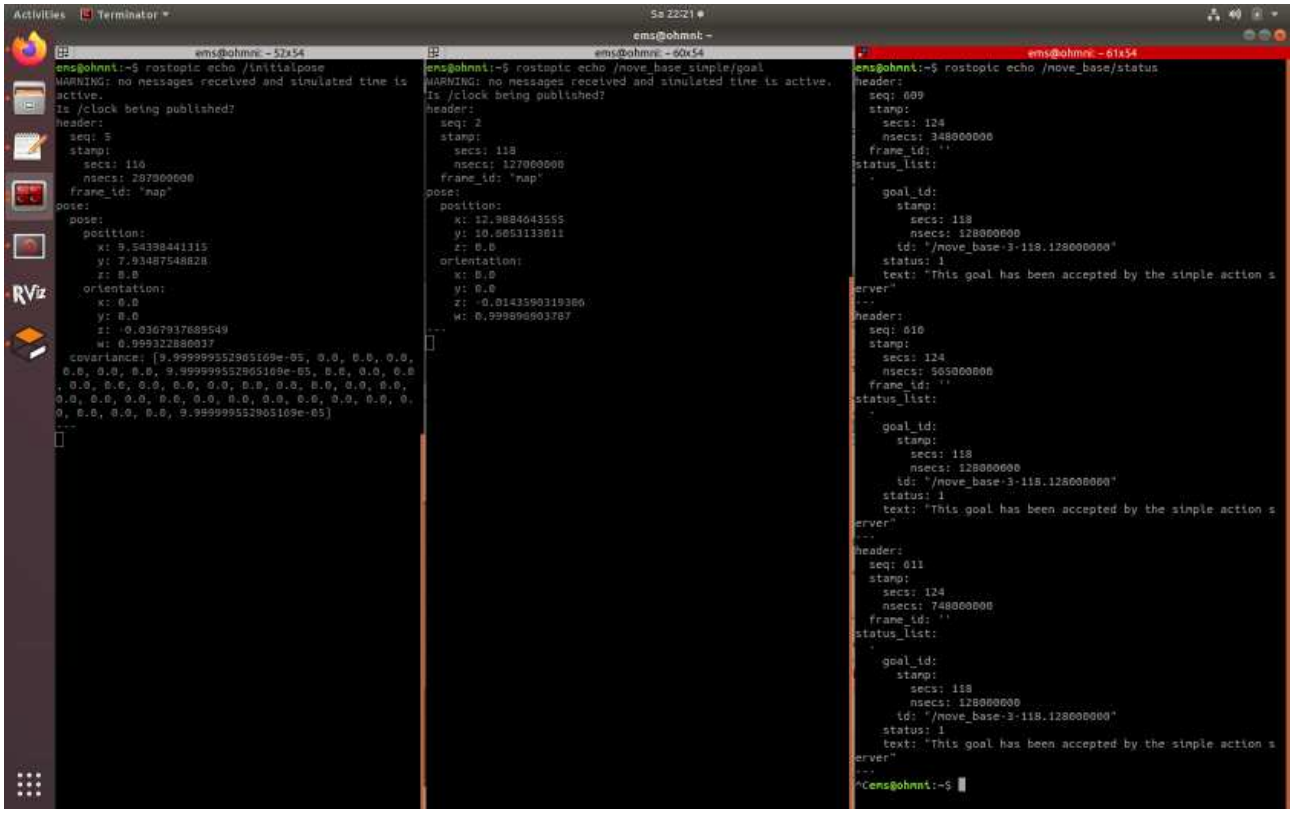

Figura 14 rostopic echo di /initialpose, /move_base_simple/goal, /move_base/status

Quindi, per cercare di risolvere il problema sono stati fatti diversi tentativi tra cui: cambiare alcuni topic che il "move_base" utilizza, modificare la mappa per farlo lavorare solo nella parte della mappa che conosce, ed anche cambiare alcuni parametri utilizzati dai file yaml che utilizza il "move_base" Questi tentativi però non hanno avuto il risultato sperato.

5.2 PROBLEMATICHE INCONTRATE

Le problematiche affrontate sono state le più svariate e queste riguardano sia problemi di codice, sia problemi esterni non dipesi da me. Tra i problemi esterni compaiono:

- L'impossibilità di accedere al Docker del robot fisico ha impedito di lavorare direttamente col robot, rendendo necessario utilizzare il codice simulato all'interno di una macchina virtuale.

- Il non poter inserire all'interno del mio computer, per un conflitto di indirizzi, una copia della macchina virtuale e dovendo quindi lavorare con il computer da remoto dell'università, il quale delle volte era spento, altre volte era usato da altri colleghi.

- Un problema all'impianto elettrico dell'università che ha bruciato l'hard disk del computer con dentro la simulazione. Sono stato così costretto, a meno di un mese dalla scadenza, a dover ricominciare tutto da capo, perdendo ulteriore tempo.

Le problematiche affrontate a livello di codice hanno richiesto una conoscenza di Ubuntu, ROS, Gazebo ed RViz che inizialmente non possedevo. Queste sono state:

- Il codice di simulazione del robot presentava dei difetti. Come, per esempio, le trasformazioni delle ruote che non erano collegate al base_link (spiegato al paragrafo 4.2.2), oppure una

mancanza di librerie che servivano a controllare la simulazione del robot, infatti le prime volte che veniva avviato, il display del robot cominciava a ruotare all'impazzata.

- Una serie di errori, risolvibili semplicemente scaricando alcuni pacchetti ROS o indicando l'indirizzo in cui il sistema poteva trovarli, il cui codice di errore nel terminale non lo indicava chiaramente. Perciò per ogni problema è stato necessario cercare dei forum che spiegassero come risolverlo.

- La sovrapposizione di due o più mappe. Difatti all'avvio della simulazione del robot si generava una mappa detta "fake_map" che andava in contrasto con la mappa caricata nel "map_server", impedendo così al robot di decidere dove fissare gli obiettivi. Legato a ciò, un altro problema è stato il conflitto tra "slam_gmappig" e "map_server". Infatti se vengono avviati entrambi, si crea un conflitto in RViz che impedisce di usarlo finché uno dei nodi non viene spento.

- Parametri non adeguati nei file di lancio yaml di "move_base.launch". Questi parametri riguardano la dimensione della costmap locale e globale, che causava una serie di warning, che impedivano al robot di tracciare il percorso.

5.3 SVILUPPI FUTURI

Il principale obiettivo futuro sarà sicuramente quello di far muovere il robot. A causa dei problemi accennati al paragrafo precedente, che mi hanno fatto perdere moltissimo tempo, non sono riuscito ad individuare e risolvere il problema finale. Tale compito spetterà a chi dopo di me dovrà confrontarsi con il problema, trovando in tale tesi una guida rapida alla comprensione del robot ed una scorciatoia nel settaggio del sistema per poter lavorare. Successivamente, si potrebbe considerare l'inserimento di una serie di codici che consentirebbero l'esecuzione di vari compiti, come accennato dal professore in uno dei nostri primi incontri. Tra questi: farlo navigare in un ambiente più complesso della semplice stanza, far sì che il robot si muova restando vicino al proprietario, dialogare con gli essere umani e monitorare il suo stato di salute.

CAPITOLO 6 – APPENDICE

Si presenteranno qui di seguito tutti i materiali necessari ad integrare le nozioni teoriche presentate all'interno della tesi. Infatti verranno presentati tutti i possibili parametri dell'Ohmni Telepresence Robot (Appendice A), i file di configurazione dei vari nodi del navigation stack (Appendice B), e l'intero procedimento di avvio del sistema (Appendice C).

A - INFORMAZIONI OHMNI TELEPRESENCE ROBOT

Questa sezione fornisce tutte le informazioni disponibili online sull'Ohmni Telepresence Robot.

A1 DIMENSIONI, STAZIONE DI RICARICA, DISPLAY

Dimensions	
Height	56 inches (142.2 cm)
Base Footprint	3.9 x 17.2 inches (35.3 x 43.7 cm)
Charging Dock	
Height	2.1 inches (5.3 cm)
Base	5.5 x 4.7 inches (14.0 x 11.9 cm)
Display	
Resolution	1280×800
Touchpoints	5 Simultaneous
Panel type	IPS

Figura 15 informazioni sul robot (pt.1)

A2 BATTERIA

Battery system	
Charge Rate	20W
Cell protection	Undervoltage, Overvoltage, Short circuit, Current limit, Cell failure
Cell Balancing	Yes
Battery chemistry	LiFePO4
Battery capacity	95Wh
Runtime (Full-load)	4-5hr
Runtime (Standby)	8-10hr
Charging Input	100-240V AC, 50/60 Hz

Figura 16 informazioni sul robot (pt.2)

A3 TELECAMERA PER LA NAVIGAZIONE

Nav camera	
Sensor Resolution	2MP
Maximum Resolution	1920×1080
Pixel Size	320×240
Chroma	RGB Bayer
Pixel size	3.0 micron
Color Filter	Streaming Resolution: Chroma RGB Bayer 3.0 micron
Sensor Size	5865 x 3276um

Figura 17 informazioni sul robot (pt.3)

A4 TELECAMERA SUPERIORE

Front camera (Supercam)	
Sensor Resolution	13MP
Maximum Resolution	4208×3120
Streaming Resolution	1280×960 (With real-time 3x Superzoom to 4K resolution)
Color Filter	Chroma RGB Bayer
Pixel Size	3.0 micron
Sensor Size	5865 x 3276 um

Figura 18 informazioni sul robot (pt.4)

A5 CPU, ALTOPARLANTE

CPU	
Processor	Intel® Pentium™ N4200
Number of cores	4 Cores
Memory	4GB
Storage	32GB eMMC
Speaker	
Speaker type	55mm Full Range Neodyne
Power	15W
Range	150 Hz ~ 20 kHz
Volume	92dB volume range, >8x louder than human speech

Figura 19 informazioni sul robot (pt.5)

A6 MICROFONO, FOLDING HINGE

Microphone	
Mic type	Custom quad-microphone array
Microphone range	20 Hz-20 kHz
DSP	Dedicated DSP chip with Echo Cancellation and Beamforming
Noise filtering	Environmental filtering
Folding hinge	
Power button	Illuminated power button, software on-off
Release	Interlocking pull-to-release

Figura 20 informazioni sul robot (pt.6)

A7 SERVO COLLO

Neck servo	
Torque (mN*m)	13
Rotation speed (RPM, no load)	1200
Encoder Type	Hall Effect Angle Sensor
Communication	Pulse Width Modulation (PWM)

Figura 21 informazioni sul robot (pt.7)

A8 SISTEMA DI GUIDA

Drive system	
Wheelbase	3-Wheel Base
Motor Type	Direct-Drive Three Phase AC brushless
Motor Power	2 x 90W
Maximum Speed	2 mph
Carrying Capacity	20 lbs
E-brake	Automatic e-braking
Stall detection	Automatic stall detection
Torque Limiting	Electronic torque limiting

Figura 22 informazioni sul robot (pt.8)

A9 WIFI, BLUETOOTH

WiFi	
Standards	IEEE802.11 ac/a/b/g/n
Encryption	WPA/WPA-PSK, WPA2/WPA2-PSK, WPA-Enterprise (802.11x)
Range	Far-field range, optimized for large environments
WiFi	Wi-Fi 5 (Dual-band)
Bluetooth	
Integrated Bluetooth 5.1	

Figura 23 informazioni sul robot (pt.9)

B – FILE UTILIZZATI

In questa sezione verranno riportati tutti i file di configurazione e di lancio che sono utilizzati dal sistema per il settaggio di parametri e per il lancio di nodi.

B1 MOVE_BASE

Per il nodo move_base è stato creato un launch file riportato qui di seguito

```
<!-- Run the map server -->
<node name="map_server" pkg="map_server" type="map_server" args="$(find
mappe)/mappa_statica.pgm 0.05">
</node>

  <!-- Node for mapping
  <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping">
        <param name="base_frame" value="base_link"/>
        <param name="odom_frame" value="odom"/>
        <param name="map_frame" value="map"/>
        <param name="map_update_interval" value="1.0"/>
  </node> -->

  <!--- Run AMCL -->
<include file="$(find amcl)/examples/amcl_diff.launch" />

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

```
   <rosparam file="$(find cost)/costmap_common_params.yaml" command="load"
ns="global_costmap" />
   <rosparam file="$(find cost)/costmap_common_params.yaml" command="load"
ns="local_costmap" />
   <rosparam file="$(find cost)/local_costmap_params.yaml" command="load" />
   <rosparam file="$(find cost)/global_costmap_params.yaml" command="load" />
   <rosparam file="$(find cost)/base_local_planner_params.yaml" command="load" />
   <!-- <param name="base_frame" value="base_link"/>
        <param name="odom_frame" value="tb_sim/odom"/>
        <param name="map_frame" value="map"/>
        <param name="cmd_vel_topic" value="cmd_vel"/>
</node>

</launch>
```

Vengono lanciati 3 diversi nodi:

1. Il nodo map_server che è responsabile del caricamento della mappa dell'ambiente in cui il robot si muoverà.
2. Il nodo Amcl utilizza un file di configurazione situato nel percorso relativo a ROS anziché nel catkin workspace, come indicato dal codice. Verrà presentato nell'appendice B2.
3. Move_base. Di seguito sono riportati i file di configurazione utilizzati. Per le spiegazioni dettagliata dei file, tornate al paragrafo 4.3.2.

Essi sono:

- **costmap_common_params.yaml:**

```
obstacle_range: 2.5
raytrace_range: 3
footprint: [[-0.1765, -0.2185], [-0.1765, 0.2185], [0.1765, 0.2185], [0.1765, -0.2185]]
#robot_radius: 1
inflation_radius: 0.55
observation_sources: laser_scan_sensor #point_cloud_sensor
laser_scan_sensor: {sensor_frame: base_link, data_type: LaserScan, topic: scan, marking:
true, clearing: true}
#point_cloud_sensor: {sensor_frame: base_link, data_type: PointCloud, topic: scan,
marking: true, clearing: true}
```

- **global_common_params.yaml:**

```
global_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 5.0
  static_map: true
  width: 80
  height: 80
  resolution: 0.05
  rolling_window: false
```

- **local_common_params.yaml:**

```
local_costmap:
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 10.0
  height: 10.0
  resolution: 0.05
```

- **base_local_params.yaml:**

```
TrajectoryPlannerROS:
  max_vel_x: 0.8
  min_vel_x: 0.1
  max_rotational_vel: 0.8
  min_in_place_vel_theta: 0.3

  escape_vel: -0.2
  sim_time: 2.0
  path_distance_bias: 0.6
  goal_distance_bias: 0.6


  acc_lim_th: 3.2
  acc_lim_x: 2.5
  acc_lim_y: 2.5

  holonomic_robot: true
  meter_scoring: true
```

B2 AMCL

Il nodo amcl è lanciato all'interno del file move_base.launch. Si presenterà ora il file di configurazione relative:

```
<launch>
<node pkg="amcl" type="amcl" name="amcl" output="screen">
  <!-- Publish scans from best pose at a max of 10 Hz -->
  <param name="odom_model_type" value="diff"/>
  <param name="odom_alpha5" value="0"/>
  <param name="transform_tolerance" value="5.0" />
  <param name="gui_publish_rate" value="10.0"/>
  <param name="laser_max_beams" value="30"/>
  <param name="min_particles" value="500"/>
  <param name="max_particles" value="5000"/>
```

```
  <param name="kld_err" value="0.05"/>
  <param name="kld_z" value="0.99"/>
  <param name="odom_alpha1" value="2"/>
  <param name="odom_alpha2" value="2"/>
  <!-- translation std dev, m -->
  <param name="odom_alpha3" value="2"/>
  <param name="odom_alpha4" value="2"/>
  <param name="laser_z_hit" value="0.5"/>
  <param name="laser_z_short" value="0.05"/>
  <param name="laser_z_max" value="0.05"/>
  <param name="laser_z_rand" value="0.5"/>
  <param name="laser_sigma_hit" value="0.2"/>
  <param name="laser_lambda_short" value="0.1"/>
  <param name="laser_lambda_short" value="0.1"/>
  <param name="laser_model_type" value="likelihood_field"/>
  <!-- <param name="laser_model_type" value="beam"/> -->
  <param name="laser_likelihood_max_dist" value="2.0"/>
  <param name="update_min_d" value="0.2"/>
  <param name="update_min_a" value="0.5"/>
  <param name="odom_frame_id" value="odom"/>
  <param name="resample_interval" value="1"/>
  <param name="transform_tolerance" value="0.1"/>
  <param name="recovery_alpha_slow" value="0.0"/>
  <param name="recovery_alpha_fast" value="0.0"/>
</node>
</launch>
```

B3 LAUNCH_TB2.LAUNCH

Questo file di lancio è responsabile dell'avvio della simulazione del robot in Gazebo. Tale file è stato creato per accelerare il processo di avvio della simulazione. Esso lancerà un altro file di lancio, "empty.launch", il quale ne lancerà altri due.

- **Launch_tb2.launch:**

```
<launch>
<include file="$(find gazebo_environment)/launch/empty.launch" />
</launch>
```

- **empty.launch:**

```
<?xml version="1.0"?>
<launch>
  <arg name="debug" default="false"/>
  <arg name="scenario" default="empty"/>
  <arg name="extra_gazebo_args" default=""/>
  <arg name="robot_ns" default="tb_sim"/>
  <arg name="use_external_controller" default="false"/>
  <arg name="use_joystick" default="false"/>
  <arg name="enable_teleop" default="true"/>
  <arg name="enable_rviz" default="true"/>
```

```
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>

  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="paused" value="false"/>
    <arg name="use_sim_time" value="$(arg use_sim_time)"/>
    <arg name="gui" value="$(arg gui)"/>
    <arg name="headless" value="$(arg headless)"/>
    <arg name="debug" value="$(arg debug)"/>
    <arg name="extra_gazebo_args" value="$(arg extra_gazebo_args) --verbose"/>
  </include>

  <include file="$(find tb_gazebo)/launch/spawn_tb.launch">
    <arg name="x" value="0.0"/>
    <arg name="y" value="0.0"/>
    <arg name="z" value="0.1"/>
    <arg name="robot_ns" value="$(arg robot_ns)"/>
    <arg name="enable_rviz" value="$(arg enable_rviz)"/>
  </include>

  <group if="$(arg enable_teleop)">
    <include file="$(find tb_teleop)/launch/manual_controller.launch">
      <arg name="use_external_controller" value="$(arg use_external_controller)"/>
      <arg name="use_joystick" value="$(arg use_joystick)"/>
    </include>
  </group>

 <!-- fake map frame for development
  <node pkg="tf2_ros" type="static_transform_publisher" name="fake_map"
    args="0.0 0.0  0.0 0 0 0 map odom" /> -->

 <node pkg="gmapping" type="slam_gmapping" name="slam_gmapping">
  <param name="base_frame" value="base_link"/>
  <param name="odom_frame" value="odom"/>
  <param name="map_frame" value="map"/>
  <param name="map_update_interval" value="1.0"/>
 </node>

 </launch>
```

- **spawn_tb.launch:**

```
 <?xml version="1.0"?>
 <launch>
  <!-- TODO solving robot namespace to add multiple robot -->
  <arg name="robot_ns" default="tb_sim1"/>
  <arg name="verbose" value="true" />
  <arg name="x" default="0.0"/>
  <arg name="y" default="0.0"/>
  <arg name="z" default="0.3"/>
```

```
  <arg name="yaw" default="0.0"/>
  <arg name="enable_rviz" default="true"/>

  <!-- <node
    name="tf_footprint_base"
    pkg="tf"
    type="static_transform_publisher"
    args="0 0 0 0 0 0 base_link base_footprint 40" /> -->
    <!-- static_transform_publisher x y z yaw pitch roll frame_id child_frame_id
period_in_ms -->
  <param
    name="robot_description"
    command="$(find xacro)/xacro '$(find tb_description)/urdf/tb_sim.urdf.xacro' --inorder"
/>
  <node
    name="spawn_model"
    pkg="gazebo_ros"
    type="spawn_model"
    output="screen"
    args="-urdf
          -param robot_description
          -model tb_sim
          -x $(arg x)
          -y $(arg y)
          -z $(arg z)
          -Y $(arg yaw)"/>
  <node
    name="joint_state_publisher"
    pkg="joint_state_publisher"
    type="joint_state_publisher" />
  <node
    name="robot_state_publisher"
    pkg="robot_state_publisher"
    type="robot_state_publisher" />

  <group if="$(arg enable_rviz)">
    <node
      name="rviz"
      pkg="rviz"
      type="rviz"
      args="-d $(find tb_gazebo)/rviz/gazebo.rviz" />
   </group>
</launch>
```

- **manual_controller.launch:**

```
<?xml version="1.0"?>
<launch>

  <arg name="use_external_controller" default="true"/>
  <arg name="use_joystick" default="false"/>
```

```
  <group if="$(arg use_external_controller)">
    <group if="$(arg use_joystick)">
      <!-- TODO: define joystick control -->
    </group>

    <group unless="$(arg use_joystick)">
      <!--use keyboard: require install http://wiki.ros.org/teleop_twist_keyboard-->
      <node name="teleop_keyboard" pkg="teleop_twist_keyboard"
type="teleop_twist_keyboard.py"
       output="screen">
        <remap from="/cmd_vel" to="/tb_sim/cmd_vel"/>
      </node>
    </group>
  </group>

  <group unless="$(arg use_external_controller)">
    <!-- Add topic name in GUI to control the robot: /tb_sim/cmd_vel -->
    <node name="robot_steering" pkg="rqt_robot_steering" type="rqt_robot_steering">
    </node>
  </group>
</launch>
```

C – PROCEDIMENTO PER L'AVVIO

In questo paragrafo verranno presentati tutti i comandi da lanciare e i passaggi da seguire affinché il sistema sia pronto all'utilizzo.

- **Avvio simulazione robot.**

  Aprire un terminale e digitare i seguenti comandi:

  ```
  cd tb-simulation
  cd ros_ws
  cd launch
  chmod +x Launch_tb2.launch
  roslaunch ./Launch_tb2.launch
  ```

- **Creazione di un mondo chiuso in Gazebo.**

  Cliccare in alto a destra su "Edit".
  Cliccare su "Building Editor".

  Ora è possibile creare la stanza in Gazebo inserendo pareti, scale, porte ed è possibile importare altri elementi scaricati. Poi è necessario:

  Cliccare su "File".
  Cliccare su "Exit Building Editor".
  Cliccare su "Save and Exit".
  Scegliere un nome.
  Cliccare su "Save".

- **Muovere il robot.**

  È necessario adesso che il robot si muova nell'ambiente. Usando "rqt_robot_steering", impostante il comando "/tb_cmd_vel" e assegnate una velocità lineare e una angolare per farlo muovere. Se dovesse cadere sarà necessario:

  Cliccare su "Edit"
  Cliccare su "Reset Model Poses"

  Usando RViz è possibile notare se il robot ha scansionato tutta la mappa. A quel punto è possibile fermare il robot e salvare la mappa.

- **Salvare la mappa.**

  Aprire un terminale e digitare il seguente comando:

  rosrun map_server map_saver -f <nome_mappa>

  È necessario sostituire <nome_mappa> con il nome che le si vuole dare. Dopo aver seguito il comando, nella home, appariranno 2 nuovi file. Tagliateli e metteteli nella cartella "Mappe". Ora è necessario aprire, con un editor di testo, il move_base.launch e sostituire il nome della vecchia mappa con il nuovo nome assegnatoli.

- **Fermare slam_gmapping.**

  Aprire un terminale e digitare il seguente comando:

  rosnode kill /slam_gmapping

- **Avvio move_base.launch.**

  Aprire un terminale e digitare i seguenti comandi:

  cd navigation
  cd ws_navigation
  cd launch
  chmod +x move_base.launch
  roslaunch ./move_base.launch

- **Aggiunta display in RViz.**

  Se è la prima volta che avviate il move_base.launch, sarà necessario inserire i display e modificare i topic come mostrato al paragrafo 4.4.
  Se non lo è, RViz sarà già disponibile all'uso del navigation stack.

Terminata tale procedura, è possibile far muovere il robot tramite "2D Pose Estimate" e "2D Nav Goal", come indicato nel paragrafo 4.4.2

BIBLIOGRAFIA E SITOGRAFIA

[1] L. Cavanini. Integrazione e sviluppo di algoritmi di localizzazione per smart wheelchair, 2012.

[2] G. Corredani Sviluppo di algoritmi di localizzazione e obstacle avoidance per dispositivi assistivi mobili in ambienti indoor.

[3] https://wiki.ros.org/it

[4] https://docs.ohmnilabs.com/

[5] https://dockertutorial.it/

[6] https://ohmnilabs.com/

[7] https://answers.ros.org/questions/

[8] https://en.wikipedia.org/wiki/Main_Page