
RELAZIONE BORSA DI STUDIO

**Progetto di servizi innovativi per l'uso
efficiente ed in sicurezza delle risorse elettriche
e per l'Assisted Living**

Andrea Efficace

Università degli Studi di Roma - Tor Vergata

- Da 17 febbraio, al 22 maggio 2023 -

- Dal 15 giugno, al 11 ottobre 2023-

11 ottobre 2023

Indice

1 Introduzione	5
1.1 Le Residenze Sanitarie Assistenziali	5
1.2 Differenze con Ospedali e Case di Cura	5
1.3 La Robotica nelle Strutture Assistenziali	6
2 Il Robot Pepper	7
2.1 Panoramica delle Funzionalità di Base	7
2.2 Specifiche Tecniche	8
2.3 L'Ambiente di Sviluppo Proprietario - NAOqiSDK	20
3 Architettura Software ed Ambiente di Sviluppo	26
3.1 ROS - Robot Operating System	26
3.2 ROS File System	27
3.3 ROS Computation Graph	28
3.4 Docker e l'integrazione con ROS	29
4 Obiettivi Raggiunti (al 22-05-2023)	31
5 Evoluzione dell'architettura: da ROS1 a ROS2	34
5.1 Stato dello Sviluppo in ROS2	35
5.2 Principali Funzionalità Perdute	35
6 SLAM	38
6.1 Fondamentali ed Approcci	38
6.2 Real Time Appearance-Based Mapping	39
7 Navigazione Autonoma	43
7.1 NAV2	43
7.2 Il Funzionamento	45
7.3 Mappatura e Rappresentazione dell'Ambiente	47
7.4 Stato dell'Implementazione	49

8 Manuale Utente	52
8.1 L'Ambiente di Sviluppo	52
8.2 Struttura del Container	53
8.3 Script Principali e Flusso di lavoro	53
8.4 Il pacchetto PepperMeta	54
8.5 Dettagli sull'utilizzo di NAOqi SDK tramite ROS	57
9 Sviluppi Futuri	61

Abstract

La presente ricerca si concentra sulla programmazione di un robot Pepper, prodotto dalla Softbank Robotics (ex Aldebaran), per sviluppare comportamenti di navigazione autonoma e assistenza al personale qualificato nelle strutture sanitarie, come le Residenze Sanitarie Assistenziali (RSA). L'obiettivo è fornire un supporto efficace ed efficiente alle attività quotidiane dei professionisti sanitari, migliorando la qualità dell'assistenza fornita agli anziani ospiti delle RSA.

Il robot Pepper rappresenta una soluzione promettente per migliorare l'assistenza sanitaria, grazie alla sua capacità di interazione sociale, percezione ambientale e movimento autonomo. La ricerca si focalizza sulla programmazione di comportamenti di navigazione che consentono al robot di muoversi in modo sicuro e fluido all'interno delle strutture sanitarie, evitando ostacoli e adattandosi a un ambiente in costante cambiamento.

Inoltre, si vuole sviluppare un sistema di assistenza che consente al robot di collaborare con il personale qualificato nelle attività quotidiane di cura, come la somministrazione di farmaci, il monitoraggio dei parametri vitali e la gestione delle informazioni cliniche. Questo approccio permette di alleggerire il carico di lavoro degli operatori sanitari e migliorare l'efficienza complessiva delle strutture sanitarie.

L'implementazione di tali comportamenti di navigazione autonoma e assistenza richiede un'accurata progettazione dell'architettura software del robot Pepper, integrando algoritmi di localizzazione, mappatura, percezione dell'ambiente e pianificazione del movimento in ambiente non noto. A tal fine è stato utilizzato il framework offerto dal Robot Operating System[1], che rappresente lo standard de-facto nella progettazione di sistemi robotici moderni.

I risultati attesi di questa ricerca possono aprire nuove prospettive nell'applicazione dei robot nell'ambito della sanità, migliorando l'assistenza agli anziani e riducendo il carico di lavoro del personale qualificato.

1 Introduzione

1.1 Le Residenze Sanitarie Assistenziali

Le strutture RSA (Residenze Sanitarie Assistenziali) sono luoghi di accoglienza residenziale per anziani che hanno bisogno di cure e assistenza a tempo pieno a causa di problemi di salute o di autonomia compromessa. Le RSA sono spesso destinate a persone anziane che non sono più in grado di vivere autonomamente a casa o che richiedono cure e attenzioni costanti o più semplicemente di supporto per le attività quotidiane, come il vestirsi, soddisfare i propri bisogni fisiologici, la somministrazione dei farmaci e la gestione delle condizioni mediche. Le strutture RSA sono solitamente gestite da personale qualificato, come infermieri e operatori sanitari, che forniscono assistenza medica e supervisione costante necessaria.

1.2 Differenze con Ospedali e Case di Cura

RSA, ospedali e case di cura differiscono per molti aspetti, come l'obiettivo e la tipologia di assistenza, la durata del soggiorno e la conformazione strutturale dell'ambiente in cui i pazienti trascorrono la loro degenza.

Le RSA sono infatti progettate per fornire assistenza a lungo termine a persone che non necessitano di cure farmacologiche intensive o di macchinari specializzati. L'obiettivo principale delle RSA è quello di fornire un ambiente residenziale sicuro e confortevole, con un'attenzione specifica alle esigenze di assistenza quotidiana degli anziani e al peso che comporta la degenza a lungo termine sulla psiche dei pazienti. Gli ospedali, al contrario, sono strutture mediche specializzate che offrono cure acute e trattamenti medici intensivi che si svolgono nel più breve tempo possibile. Le case di cura, invece, sono simili alle RSA ma solitamente si rivolgono a persone con esigenze di assistenza più complesse e impegnative.

Una grande differenza è inoltre data dall'ambiente e dalle attività che i pazienti svolgono in queste strutture: Gli ospedali sono ambienti clinici e altamente strutturati, in cui si presta particolare attenzione alla cura medica e alla guarigione. Le RSA,

invece, cercano di creare un ambiente domestico e familiare, con attenzione alla qualità della vita degli anziani. Le RSA possono offrire attività ricreative, servizi di ristorazione, programmi sociali e supporto psicologico per favorire il benessere e l'inclusione sociale degli anziani ospiti.

1.3 La Robotica nelle Strutture Assistenziali

La combinazione di robotica e domotica può apportare un valore aggiunto significativo nei contesti medici assistenziali delle RSA (Residenze Sanitarie Assistenziali). La robotica può contribuire a migliorare l'assistenza fornita ai pazienti, aumentando l'efficienza, la sicurezza e la qualità della cura. I robot possono svolgere una varietà di compiti ripetitivi, come l'assistenza nella somministrazione dei farmaci, il monitoraggio dei parametri vitali come pressione sanguigna, temperatura e ossigenazione del sangue. Possono risultare un supporto valido anche nei contesti riabilitativi e in quelli ludici o per il controllo costante di quei pazienti con problematiche più importanti e per il quale è necessario un monitoraggio continuo della loro posizione all'interno della struttura e della loro condizione.

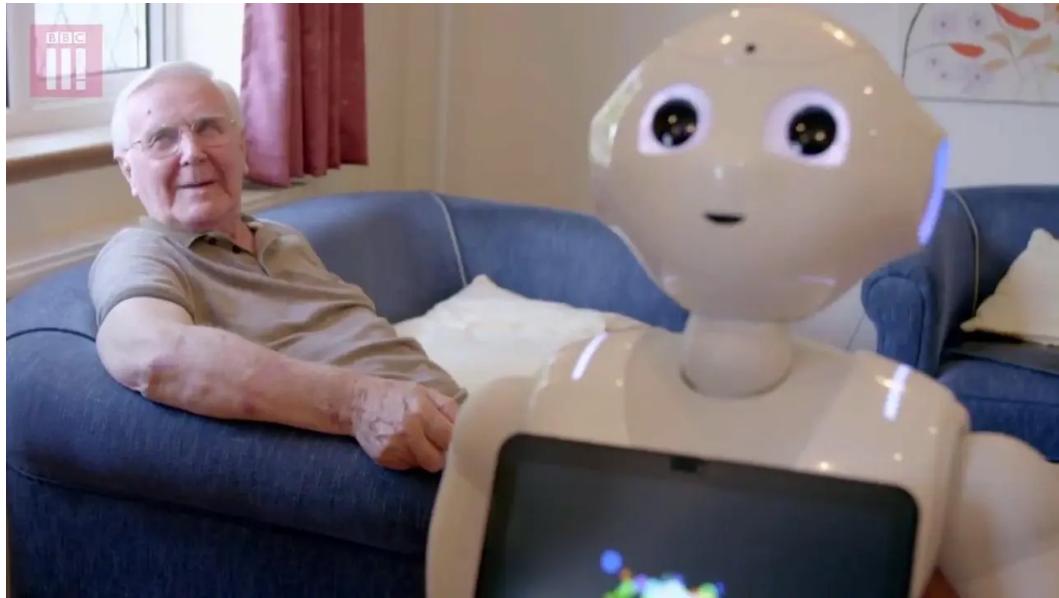


Figura 1: Il robot Pepper insieme ad un paziente in RSA.

2 Il Robot Pepper

Il robot Pepper, prodotto dalla Softbank Robotics (ex Aldebaran Robotics), è un robot umanoide progettato per interagire con le persone e fornire supporto nelle attività quotidiane. Il suo aspetto umanoide, con un volto dotato di occhi e bocca, e una stazza simile a quella di un bambino o di un giovane adulto fornisce al robot un'aria amichevole, il che lo rende ottimo per il contesto assistenziale nel quale la ricerca si vuole concentrare.

2.1 Panoramica delle Funzionalità di Base

La Softbank Robotics mette a disposizione diverse funzionalità già configurate e implementate nel robot al momento dell'acquisto.

- **Interazione Sociale:** Il robot è dotato di microfoni e altoparlanti per rilevare e riprodurre il suono, oltre a telecamere e sensori di rilevamento del movimento per percepire le persone intorno a lui. Utilizzando l'intelligenza artificiale e algoritmi di elaborazione del linguaggio naturale, Pepper può riconoscere le voci umane, rispondere a domande e comandi, partecipare a conversazioni e persino riconoscere e interpretare le espressioni facciali.
- **Movimento Autonomo:** Pepper è in grado di muoversi autonomamente grazie a una base mobile a ruote. Questo gli consente di spostarsi in modo fluido all'interno di un ambiente, evitando ostacoli grazie ai suoi sensori di prossimità. Il robot può anche fare gesti e movimenti del corpo per enfatizzare le sue interazioni e comunicare in modo più efficace.
- **Sensibilità Emotiva:** Pepper è stato progettato per essere in grado di riconoscere le emozioni umane attraverso l'analisi delle espressioni facciali, della tonalità della voce e del linguaggio del corpo. Il robot può quindi adattare le sue risposte e interazioni per rispecchiare e rispondere alle emozioni delle persone con cui interagisce.

2.2 Specifiche Tecniche

Facendo riferimento alla documentazione fornita sul sito del produttore, forniamo una panoramica sulle specifiche tecniche e sulla sensoristica a bordo del robot.

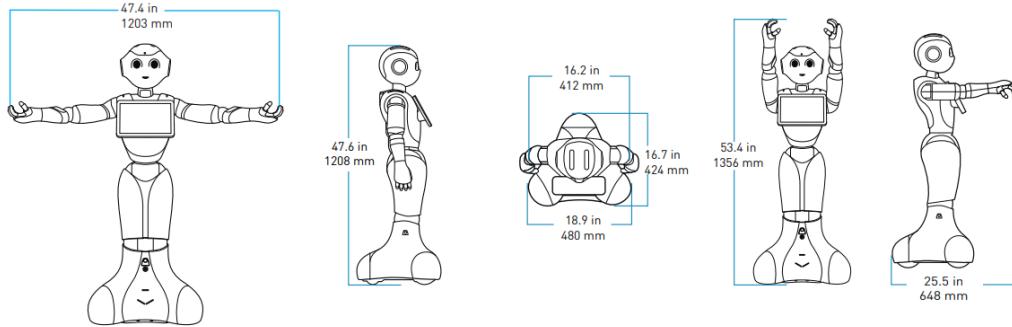


Figura 2: Dimensioni del Robot

Informazioni Fisiche (fig. 2)	
Dimensioni (con packaging)	vedi diagramma in fig. 2 (1400 × 580 × 580)
Peso	29.6kg
Ambiente di Lavoro	
Temperatura	da 5°C a 35°C
Umidità	da 20% a 80%
Classe IP	IPX0
Condizioni di Stoccaggio	
Range di Temperatura	da 5°C a 45°C
Specifiche della Batteria	
Run time loss a 25° C	20%/anno
Self-Discharging a 45°C	3.5%/mese
Autonomia Minima	7 ore
Autonomia Tipica	12 ore
Autonomia Massima	20 ore
Tempo di ricarica da 0% (approx.)	8h e 20min da 0 a 100%
Tipologia	Celle cilindriche agli ioni di litio (NMC)
Voltaggio Nominale	26.46 V
Voltaggio Minimo	26.45 V
Voltaggio massimo di carica	29.4 V
Corrente di ricarica	8 A
Capacità Tipica	30 Ah
Temperatura di Lavoro in Ricarica	da 10 a 35 °C

Tabella 1: Specifiche Generali

On-Board Computer	
Processore	<i>Intel ATOM® E3845 Formerly Bay Trail</i>
Tipologia	<i>Quad Core</i>
Clock Speed	<i>1.91 Ghz</i>
Adjusted Peak Performance (APP)	<i>0.00344 WT</i>
RAM	<i>4 GB DDR3</i>
Flash Memory	<i>32 GB eMMC</i>

Tabella 2: Specifiche dell'on-board computer

Gradi di Libertà (Testa) (fig. 3)	
Yaw	$-119.5^\circ - +119.5^\circ (-2.09 - +2.09 \text{ rad})$
Pitch	$-40.5^\circ - +25.5^\circ (-0.71 - +0.45 \text{ rad})$
Gradi di Libertà (Braccia) (fig. 4)	
Pitch (spalla)	$-119.5^\circ - +119.5^\circ (-2.09 - +2.09 \text{ rad})$
Roll (spalla)	$+0.5^\circ - +89.5^\circ (-0.01 - +1.56 \text{ rad})$
Yaw (gomito)	$-119.5^\circ - +119.5^\circ (-2.09 - +2.09 \text{ rad})$
Roll (gomito)	$-89.5^\circ - -0.5^\circ (-1.56 - -0.01 \text{ rad})$
Gradi di Libertà (Mani) fig. 5	
Yaw	$-104.5^\circ - +104.5^\circ (-1.82 - +1.82 \text{ rad})$

Tabella 3: Gradi di Libertà per i giunti superiori.

Gradi di Libertà (Gambe) (fig. 6)	
Roll (bacino)	$-29.5^\circ - +29.5^\circ (-0.51 - +0.51 \text{ rad})$
Pitch (bacino)	$-59.5^\circ - +59.5^\circ (-1.04 - +1.04 \text{ rad})$
Pitch (ginocchia)	$-29.5^\circ - +29.5^\circ (-0.51 - +0.51 \text{ rad})$
Gradi di Libertà (Base) (fig. 7)	
Ruota FL (frontale SX)	<i>Base mobile omnidirezionale su tutte e tre le ruote</i>
Ruota FR (frontale RX)	<i>con velocità massima di 2km/h</i>
Ruota B (posteriore)	<i>e pendenza massima di 5°</i>

Tabella 4: Gradi di Libertà per i giunti inferiori.

Tablet	
Dimensioni	246 x 175 x 14.5 mm
CPU	1.3GHz quad-core ARM Cortex-A7
Adjusted Peak Performance (APP)	0.003156 WT
DDR3 SDRAM	1 GB
Flash Memory	32 GB eMMC
LCD	IPS, 1280 x 800, 24 bit true colour
Touchscreen	Capacitivo, multi-touch a 5 punti
Camera	2 megapixels
Sistema Operativo	Android

Tabella 5: Specifiche Tablet

Speaker (fig. 8)	
Posizione	Un altoparlante per orecchio (A-B)
Impedenza	8Ω
Max. SPL	74 dB/W/m
Frequenza	da 400Hz a 9kHz (-6dB)
Potenza d'Uscita	7 W RMS
Microfoni (fig. 8)	
Posizione	Quattro microfoni sopra la testa (A-B-C-D)
Sensitività	-12 dBV (0.71 Vpp) @1Khz
Max. SPL	110 Db
Tipo	Omnidirezionale

Tabella 6: Specifiche Audio

Flat Imaging 2D (fig. 9)

Posizione	Una telecamera sulla fronte e sulla bocca
Modello	OV5640
Tipo	SOC Image Sensor
Risoluzione	5 Megapixels
Dimensione	1/4 inch
Active Pixels	2592 x 1944
Pixel Size	1.4 x 1.4 μm
Range Dinamico	64 dB @8x gain
Signal/Noise Ratio	36 dB (max)
Responsività	600 mV/Lux-sec
Shutter Type	Rolling shutter / frame exposure
Camera Output	640x480 @30/15 fps
Field of View	54.4° HFOV, 44.6° VFOV
Tipo di Fuoco	Autofocus, da 10cm fino a ∞

Depth e Stereo Imaging (fig. 10)

Posizione	Due telecamere posizionate dietro gli occhi
Modello	OV4689
Tipo	CMOS Image Sensor
Dimensione	1/3 inch
Active Pixels	1280 x 720
Shutter Type	Rolling shutter / frame exposure
Camera output	2560 x 720 @15 fps
Tipo di Fuoco	Fuoco fisso, 40 cm to ∞
Field of View	90.6° HFOV, 56.3° VFOV
Sensore 3D	57.2° HFOV, 44.3° VFOV

Tabella 7: Specifiche Video

Bottoni (fig. 11, fig. 12)

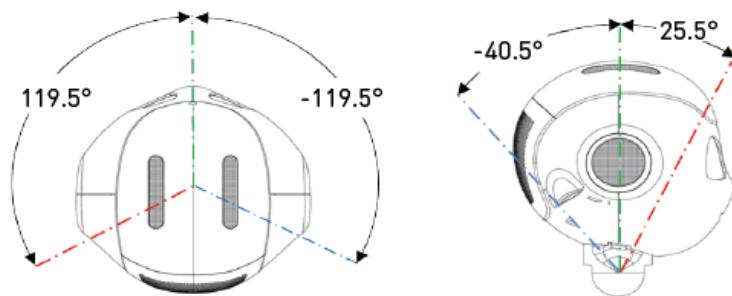
Bottone A (on/off)	Sul petto, dietro il tablet
Bottone B (stop)	Dietro il collo
Wheel Bumpers	Due davanti (A,B) , uno sul retro della base (C)

Tabella 8: Posizione dei Bottoni

Connettività

Wi-Fi	Modulo WNC DHXA222-802.11
Standard supportato	802.11 a/b/g/n
Ethernet	Supportato, solo per manutenzione
Bluetooth	4.0 a basso consumo

Tabella 9: Supporto alla connettività



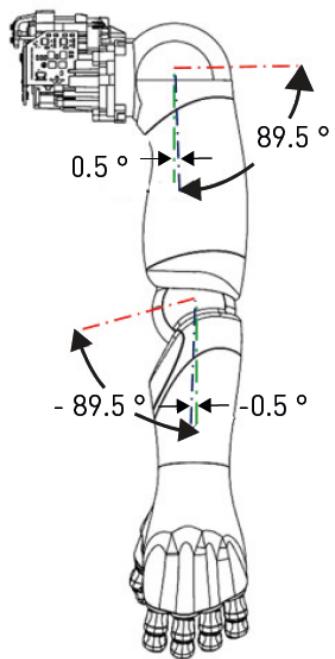
HeadYaw

HeadPitch

Figura 3: Gradi di Libertà della Testa

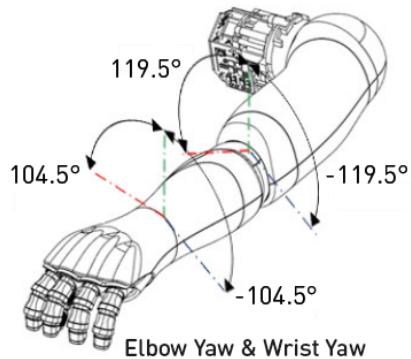
Laser Puntuali (fig. 13)	
Posizione	Un laser posizionato sotto ogni lato della base (D-E-F) Tre laser posizionati sul lato frontale della base (A-B-C)
Classe	1M
Lunghezza d’Onda	808 nm
Modalità di Funzionamento	Pulsata
Frame Rate	6.25 Hz
Laserscan (fig. 14)	
Posizione	Uno scan frontale (A), due scan per lato (B-C)
Lunghezza d’Onda	808 nm
Modalità di Funzionamento	Pulsata
Frame Rate	6.25 Hz
Cono di visione	60° (per ogni laser)
Sensori Infrarossi (fig. 15)	
Posizione	Due sensori ad ogni lato della base (A-B)
Lunghezza d’Onda	940nm
Distanza	da 0 a 50cm ad un’altezza di 27cm dal suolo
Angolo	4°
Sonar (fig. 16)	
Posizione	Un sonar frontale (A) ed uno posteriore (B)
Frequenza	42 kHz
Sensibilità	-86 dB
Risoluzione	0.03m
Range di rilevamento	da 0 a 3m. Ogni oggetto più vicino di 30cm è rilevato come fosse a 30 cm
Cono di visione	60°

Tabella 10: Sensori Ambientali



ShoulderRoll & ElbowRoll

Figura 4: Gradi di Libertà di Spalla e Gomito



Elbow Yaw & Wrist Yaw

Figura 5: Gradi di Libertà di Gomito e Polso

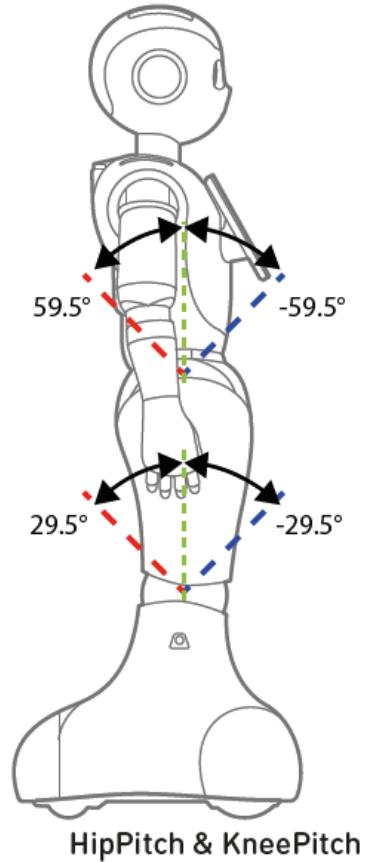


Figura 6: Gradi di Libertà di Busto e Ginocchia

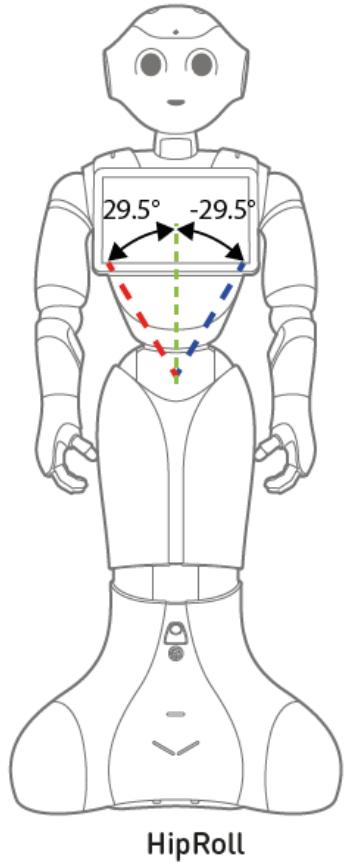


Figura 7: Gradi di Libertà del Busto

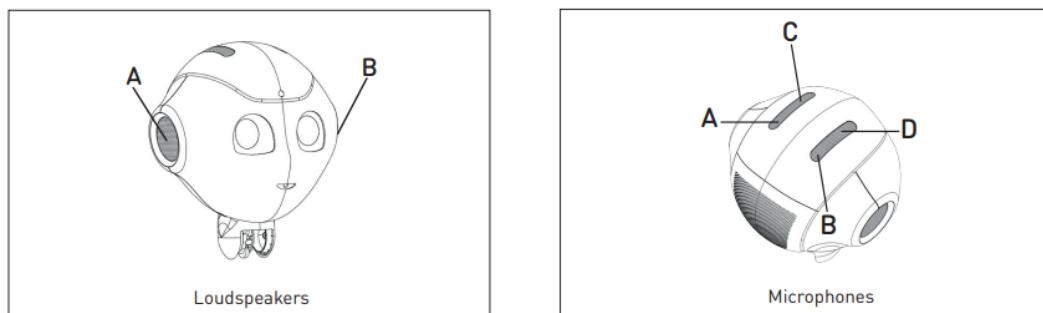


Figura 8: Speaker e Microfoni

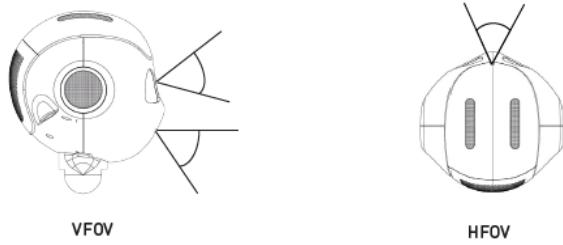


Figura 9: Campo visivo 2D

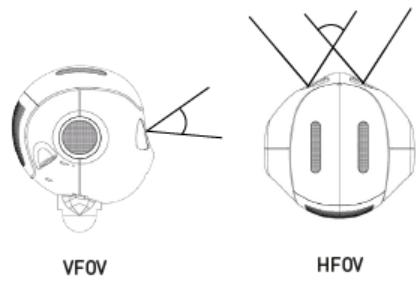


Figura 10: Depth e Stereo Camera

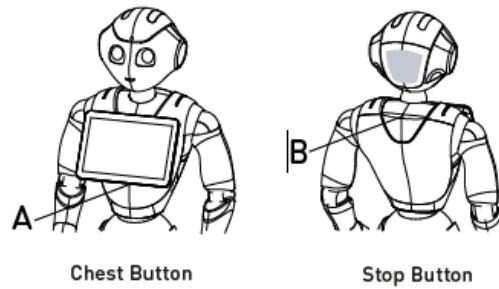


Figura 11: Bottoni

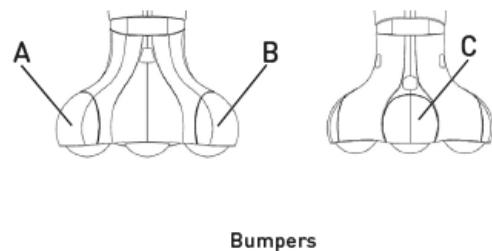


Figura 12: Bumpers

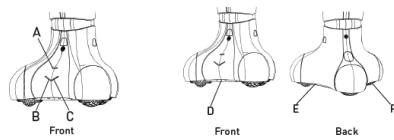


Figura 13: Laser Puntuali

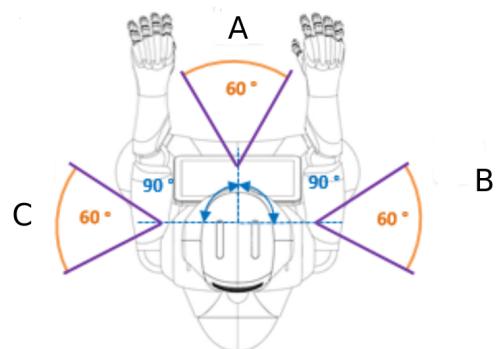


Figura 14: Laserscan

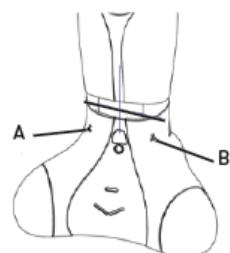


Figura 15: Sensori Infrarossi

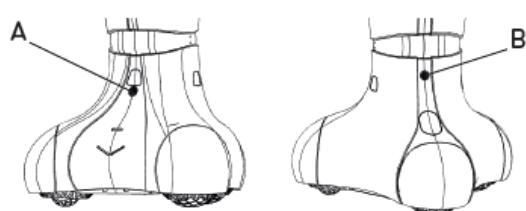


Figura 16: Sonar

Complessivamente, lo spazio dei sensori visibili al robot è quello indicato in fig. 17

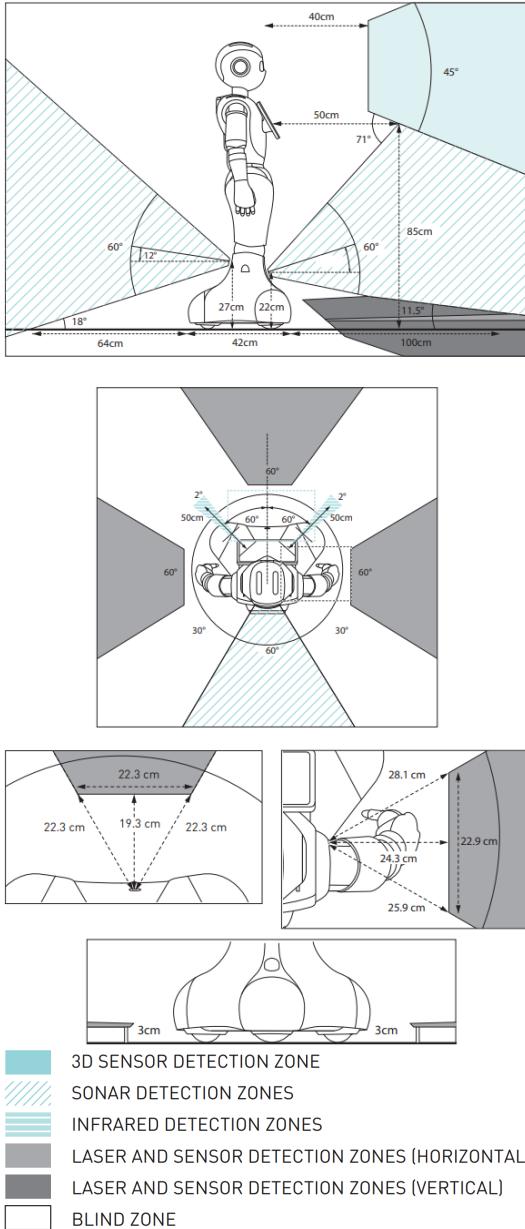


Figura 17: Spazio complessivo rilevabile dal robot

2.3 L'Ambiente di Sviluppo Proprietario - NAOqiSDK

La prima modalità d'accesso per lo sviluppatore è fornita dall'Aldaberan United Robotics Group tramite il *software development kit* NAOqi. Questo SDK è sviluppato in *Python* (v 2.7) e *C++* ed espone allo sviluppatore le funzionalità di base con cui il robot viene venduto, partendo dal livello più basso come l'accesso alle aree di memoria in cui i sensori scrivono le loro letture, fino al livello più alto come

Programming Languages	Bindings running on		Choregraphe support	
	Computer	Robot	Build Apps	Edit code
Python	✓	✓	✓	✓
C++	✓	✓	✗	✗
Java	✓	✗	✗	✗
JavaScript	✓	✓	✓	✗
ROS	✓	✗	✗	✗

✓	OK
✗	Not available

Figura 18: Linguaggi Supportati dall'SDK

la gestione dei comportamenti e delle *posture* che il robot può assumere oppure la pianificazione di traiettorie da inseguire.

Nelle versioni più recenti dell'SDK (la 2.9), anche il linguaggio Java e Javascript è stato introdotto nel supporto e l'azienda è passata allo sviluppare in ambiente Android. Viene fornito anche un ambiente di sviluppo, chiamato Choreographe (fig. 19), che permette tramite creare le proprie funzionalità attraverso programmazione basata su diagrammi di flusso. Fino alla versione 2.5.10, tutte le risorse sono

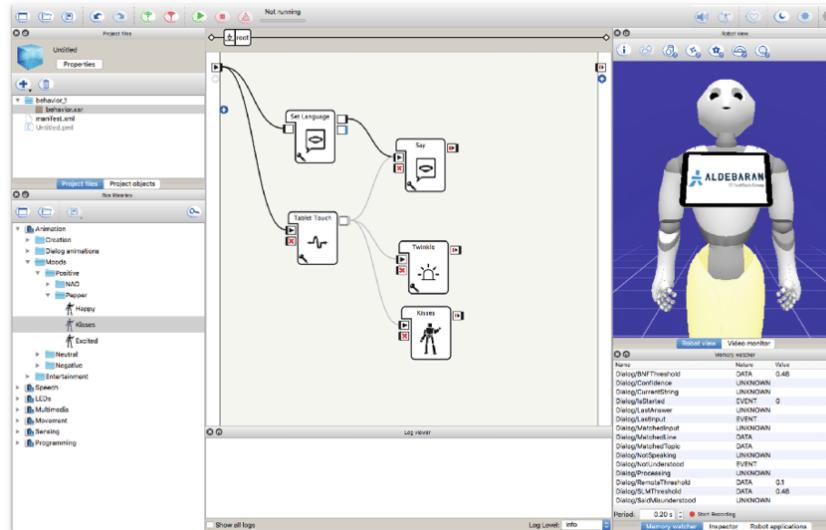


Figura 19: Screenshot della GUI di Choreographe

disponibili tramite il sito del produttore.

Lo stack software di NAOqi è basato su un'entità detta ***Broker*** che viene eseguita al momento dell'accensione del robot (eseguendo un file denominato *autoload.ini*) ed è incaricata di gestire le chiamate di avviamento per istanziare i moduli di gestione dei sensori e dei diversi motori dei giunti, mettendo ogni sottosistema in comunicazione tramite interfaccia di rete. Il modo con cui le chiamate vengono fatte sono trasparenti al programmatore in quanto si basano sulle stesse librerie messe a disposizione nelle API dell'SDK.

La struttura organizzativa è difatti un albero (fig. 20), le cui foglie sono i metodi esposti al programmatore per sviluppare funzionalità personalizzate. Le chiamate

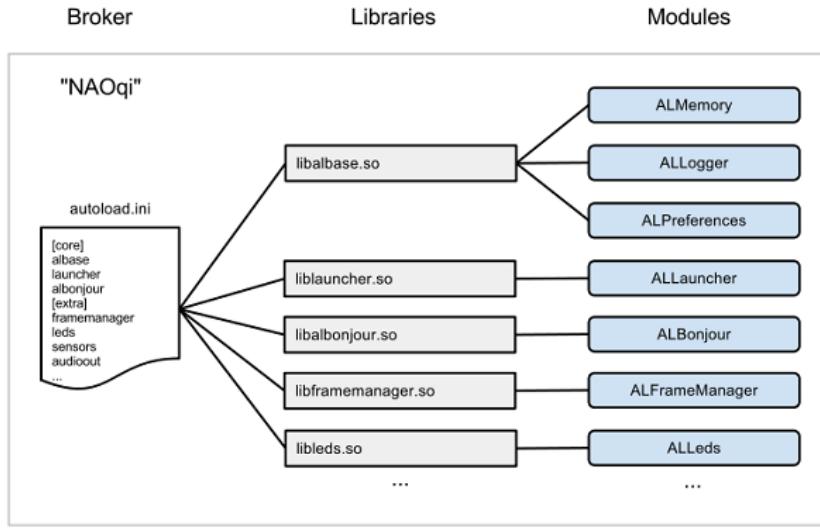


Figura 20: Stack Software di NAOqi

(fig. 21) vengono gestite da ***ALModule***, una classe responsabile di effettuare il binding tra i vari sottomoduli (come ***ALMemory*** per la gestione delle aree di memoria dei sensori o ***ALMotion*** per agire sui motori dei giunti) ed effettuare le chiamate. Il processo avviene in due modi, a seconda della modalità con cui viene eseguita la chiamata:

- **Remote-Access:** Se la chiamata avviene da remoto si compila un file eseguibile che poi viene chiamato dall'esterno. Il vantaggio di questo metodo è che

risulta più facile da testare e da correggere, ma con una perdita di prestazioni a livello di velocità di calcolo e utilizzo di memoria.

- **Local-Access:** Se la chiamata avviene in locale, il modulo rappresenta una libreria di sistema che solo il robot può usufruire. Il principale vantaggio di questo metodo è il miglioramento delle prestazioni a discapito della possibilità di manutenere il codice.

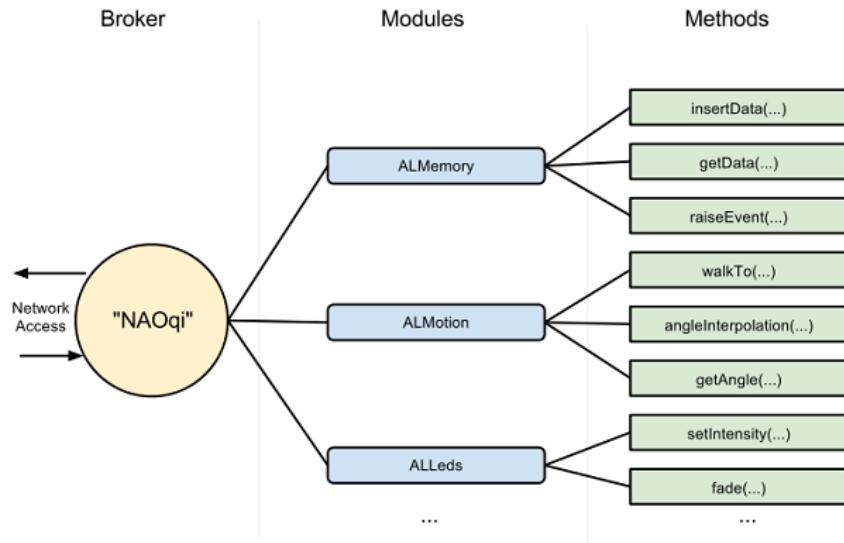


Figura 21: Schema di accesso ai moduli

Le chiamate, come spesso accade, possono essere di due tipi:

- **Bloccanti** (fig. 22): Ogni qual volta che si utilizza la segnatura `modulo.metodo()` si blocca l'esecuzione del modulo chiamante per eseguire il metodo del modulo chiamato per poi proseguire l'esecuzione del chiamante.
- **Non Bloccanti** (fig. 23): Ogni qual volta che si utilizza la segnatura `modulo.metodo()` si avvia un thread che eseguire il metodo in parallelo permettendo di eseguire più task in contemporanea. La gestione della concorrenza è demandata al modulo `ALMemory`.

Riportiamo alcuni dei moduli più importanti, gli altri sono disponibili in documentazione [2]:

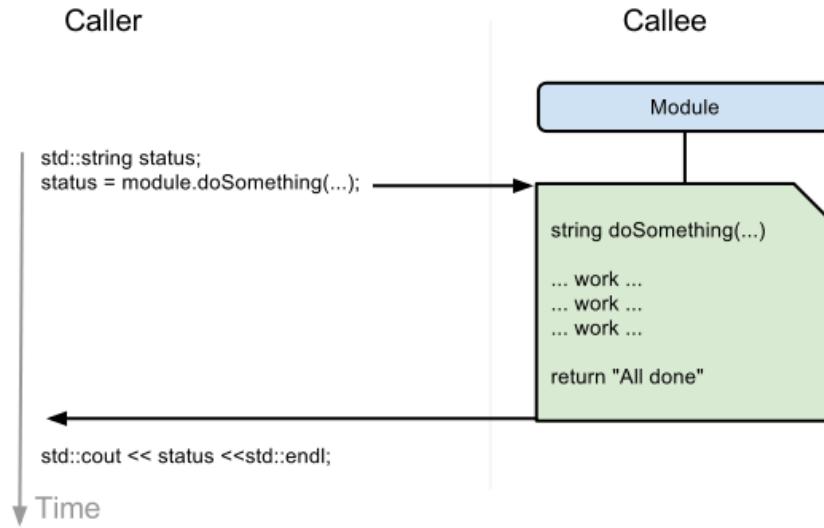


Figura 22: Schema d'esecuzione per chiamate Bloccanti

- **ALBehaviorManager:** azionamento e interruzione dei comportamenti;
- **ALConnectionManager:** gestione delle connessione alla rete e della sua configurazione;
- **ALDiagnosis:** monitoraggio dello stato del robot tramite diagnostica attive e passive;
- **ALExpressionWatcher:** gestione degli eventi in memoria al fine di generare eventi più complessi;
- **ALMemory:** gestione di ottenimento e inserimento di dati disponibili a tutti gli altri moduli;
- **ALSystem:** gestione del sistema del robot;

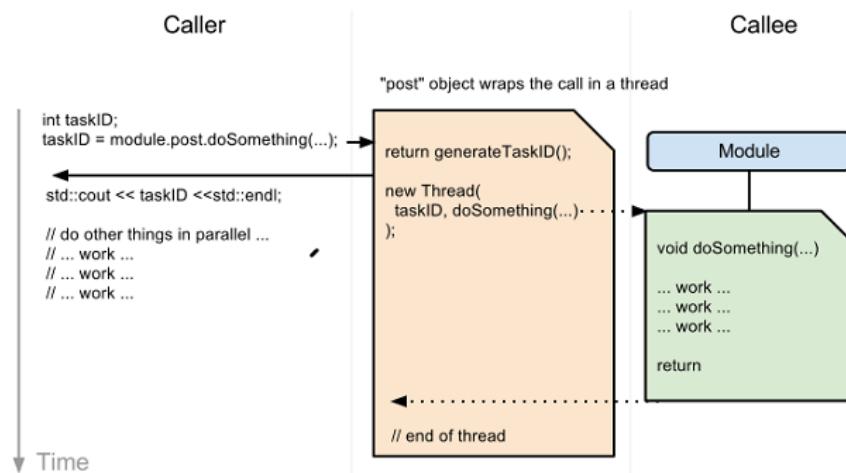


Figura 23: Schema d'esecuzione per chiamate Bloccanti

3 Architettura Software ed Ambiente di Sviluppo

L’ambiente di sviluppo messo a disposizione dalla casa madre integra una soluzione basata su ROS, il Robot Operating System, il quale permette di sfruttare C++ e Python per avere accesso al robot come se questo fosse una black-box alla quale possiamo connetterci per controllarne il funzionamento. Vediamo una panoramica del funzionamento di ROS e dell’integrazione che è stata fatta con Docker per permettere l’accesso al Pepper tramite software scritto da noi.

3.1 ROS - Robot Operating System

ROS è una piattaforma potente e flessibile per lo sviluppo di applicazioni robotiche, offrendo una vasta scelta di capacità ad alto livello, supporto per la simulazione, interoperabilità tra piattaforme e gestione modulare delle risorse. Questi vantaggi ne fanno una scelta popolare e apprezzata nell’ambiente di sviluppo di soluzioni per la robotica e l’automazione.

I suoi punti di forza riguardano diversi aspetti: In primo luogo, ROS fornisce un’ampia gamma di capacità ad alto livello, come **SLAM** (*Simultaneous Localization and Mapping*) e **AMCL** (*Adaptive Monte Carlo Localization*). Queste capacità sono distribuite attraverso pacchetti scaricabili e modulari, il che significa che possono essere utilizzate su una vasta gamma di robot senza richiedere configurazioni particolari. Questo consente agli sviluppatori di non dover affrontare ogni problema da zero, poiché molte soluzioni sono già state sviluppate e risolte.

Grazie all’enorme comunità di sviluppatori vi è anche un vasto supporto ad un’ampia gamma di sensori ed attuatori, che permettono un’astrazione degli stessi ad alto livello, permettendo di leggere il valore delle misurazioni e di inviare segnali di controllo tramite il sistema di trasporto di messaggi.

Tale sistema di trasporto è un **middleware** che si occupa di mettere in comunicazione le diverse entità del robot, che possono comunicare secondo uno standard definito dal sistema di messaggistica (che può essere ridefinito e/o espanso ad-hoc dal pro-

grammatore) e, soprattutto, che possono essere programmate in modo indipendente e anche usando diversi linguaggi per una totale flessibilità.

Tale concetto di modularità permette infatti di costruire un'architettura robusta in quanto ogni nodo della rete, ovvero ogni sottosistema, è progettato per funzionare anche se un altro sottosistema si interrompe, garantendo una maggiore resilienza e stabilità del sistema.

Infine, ROS gestisce in modo concorrente le risorse del sistema, consentendo l'esecuzione parallela di più processi e l'ottimizzazione delle risorse disponibili, che spesso sono limitate per mantenere bassi i costi di produzione e il consumo energetico.

3.2 ROS File System

La struttura del file system di ROS è organizzata in base a un approccio modulare e gerarchico, non troppo distante dai moderni file system che troviamo nei sistemi operativi più comuni (Windows, Linux ...). Questa organizzazione facilita la gestione e la condivisione dei pacchetti di software, nonché la separazione delle diverse funzionalità e componenti dei sistemi robotici. Analizziamo gli elementi in fig. 24:

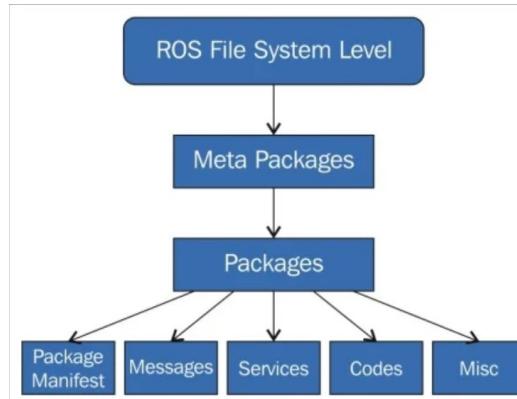


Figura 24: ROS File System

- **Meta-Packages:** usato per indicare un gruppo di pacchetti il cui utilizzo combinato è orientato alla risoluzione di uno specifico task.

- **Packages:** l’unità più semplice del file-system che raccoglie i nodi, le librerie e i file di configurazione.
- **Messages:** definiti con appositi formati nei pacchetti, costituiscono il tipo di informazione scambiato nella rete tra un nodo e l’altro.
- **Services:** un servizio è un meccanismo di comunicazione tra nodi che consente l’invio di richieste e risposte sincrone ed è definito utilizzando un messaggio di richiesta e un messaggio di risposta; Un nodo client invia una richiesta al nodo server specificando il servizio desiderato e riceve una risposta. I servizi sono utilizzati per eseguire operazioni complesse o richiedere informazioni specifiche. I servizi vengono definiti creando due file di messaggio e il codice sorgente viene generato automaticamente.
- **Package Manifests:** è un file posizionato all’interno della cartella del pacchetto che raccoglie le informazioni sul pacchetto quali l’autore, la licenza, le dipendenze e i flag di compilazione.

3.3 ROS Computation Graph

Il Computation Graph è una rappresentazione astratta delle relazioni tra i nodi e i loro topic, servizi e azioni all’interno di un sistema ROS. È un modello concettuale che permette di visualizzare e comprendere come i componenti del sistema interagiscono tra loro anche quando il loro numero cresce molto.

Come detto in precedenza, ogni nodo rappresenta un’unità di elaborazione indipendente che può essere eseguita sulla stessa macchina o anche su macchine remote connesse dalla rete. La comunicazione avviene tramite scambio di messaggi su *topic*, richieste e risposte di servizi, o tramite l’invocazione di azioni.

Abbiamo già visto a grandi linee cosa sono i servizi, vediamo ora topic ed azioni:

- **Topics:** sono canali di comunicazione asincroni che consentono ai nodi di scambiarsi dati in tempo reale. Un nodo può pubblicare dati su un topic e

altri nodi possono iscriversi a quel topic per ricevere i dati. Questo permette una comunicazione di tipo publisher-subscriber.

- **Azioni:** sono simili ai servizi, ma consentono l'esecuzione di operazioni a lunga durata con feedback intermedi. Un'azione è composta da una sequenza di stati che possono essere eseguiti in parallelo, permettendo una comunicazione bidirezionale tra il client e il server dell'azione.

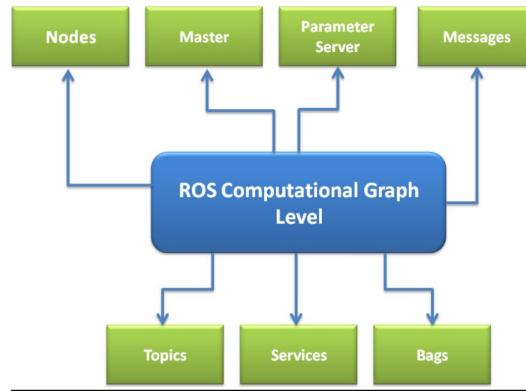


Figura 25: ROS Computation Graph

3.4 Docker e l'integrazione con ROS

Docker è una piattaforma open-source per la creazione, la distribuzione e l'esecuzione di applicazioni in ambienti isolati chiamati container. I container Docker consentono di confezionare un'applicazione e tutte le sue dipendenze in un'unità standardizzata, che può essere eseguita su qualsiasi sistema operativo che supporti Docker, senza dover gestire le complessità delle configurazioni dell'ambiente.

L'uso di Docker con ROS offre diversi vantaggi nell'ambiente di sviluppo e distribuzione delle applicazioni robotiche. Ecco una panoramica dei vantaggi principali:

- **Isolamento:** Docker consente di creare container isolati che racchiudono un'applicazione ROS e le sue dipendenze. Questo assicura che l'applicazione funzioni in modo coerente indipendentemente dall'ambiente di esecuzione, evitando conflitti tra le librerie e semplificando la gestione delle dipendenze.

- **Portabilità:** I container Docker sono indipendenti dal sistema operativo sottostante. Ciò significa che un'applicazione ROS containerizzata può essere eseguita su diversi sistemi senza dover preoccuparsi della compatibilità delle librerie o delle configurazioni specifiche del sistema.
- **Riproducibilità:** Docker permette di definire l'ambiente di esecuzione dell'applicazione in un file chiamato Dockerfile. Questo file contiene le istruzioni per creare l'immagine del container, specificando le dipendenze, le versioni delle librerie e altre configurazioni necessarie. Questa approccio consente di riprodurre facilmente l'ambiente di sviluppo su diversi sistemi e semplifica la collaborazione tra sviluppatori.
- **Scalabilità:** Docker facilita la scalabilità delle applicazioni ROS. È possibile eseguire più container, ognuno contenente un'istanza dell'applicazione ROS, distribuendo il carico di lavoro su più macchine. Questo permette di sfruttare le risorse del sistema in modo efficiente e migliorare le prestazioni complessive del sistema robotico.
- **Semplicità di distribuzione:** Grazie all'approccio containerizzato, è possibile distribuire l'applicazione ROS con tutte le sue dipendenze in un unico pacchetto. Questo semplifica la distribuzione dell'applicazione su diversi ambienti, consentendo agli utenti di eseguire l'applicazione senza dover configurare manualmente le dipendenze.
- **Ambiente di sviluppo consistente:** Utilizzando Docker come ambiente di sviluppo per ROS, tutti i membri del team di sviluppo possono lavorare su una versione coerente dell'ambiente, eliminando problemi di compatibilità tra i diversi sistemi operativi e le configurazioni locali.

4 Obiettivi Raggiunti (al 22-05-2023)

Il focus del lavoro di ricerca è stato quello di comprendere al meglio l'ambiente di sviluppo del robot e l'integrazione di ROS offerta con i pacchetti dello stack di ROS-NAOqi [3] per i robot della Aldebaran. Tramite questo insieme di pacchetti è possibile la connessione con ROS ai nodi automaticamente caricati all'accensione del Pepper, rendendo possibile l'acquisizione dei dati che i nodi pubblicano.

Allo stato attuale tramite il middleware è stato possibile costruire interfacce per la lettura e l'acquisizione di tutti i sensori a cui è agganciato un nodo e visualizzarli su RViz, un software integrato nell'insieme dei pacchetti di ROS dedito proprio alla visualizzazione a schermo di ciò che il robot e i suoi sensori stanno captando dall'ambiente.

Inoltre, per una corretta visualizzazione ed utilizzo è stato necessario effettuare modifiche al codice sorgente delle librerie offerte, in quanto non più manutenute ed ormai deprecate quasi completamente. In particolare:

- E' stato modificato il file di configurazione della telecamera di profondità per consentire il corretto caricamento dei parametri di distorsione delle lenti.
- Disabilitazione dei componenti audio del robot in quanto provocano un crash del sistema di NAOqi e non consentono la connessione con ROS.
- Creazione di diversi file di avvio personalizzati per effettuare i task di caricamento, SLAM e navigazione.

Possiamo vedere un esempio di una mappa ricostruita e di come appare il robot in RViz nelle figure a seguire.

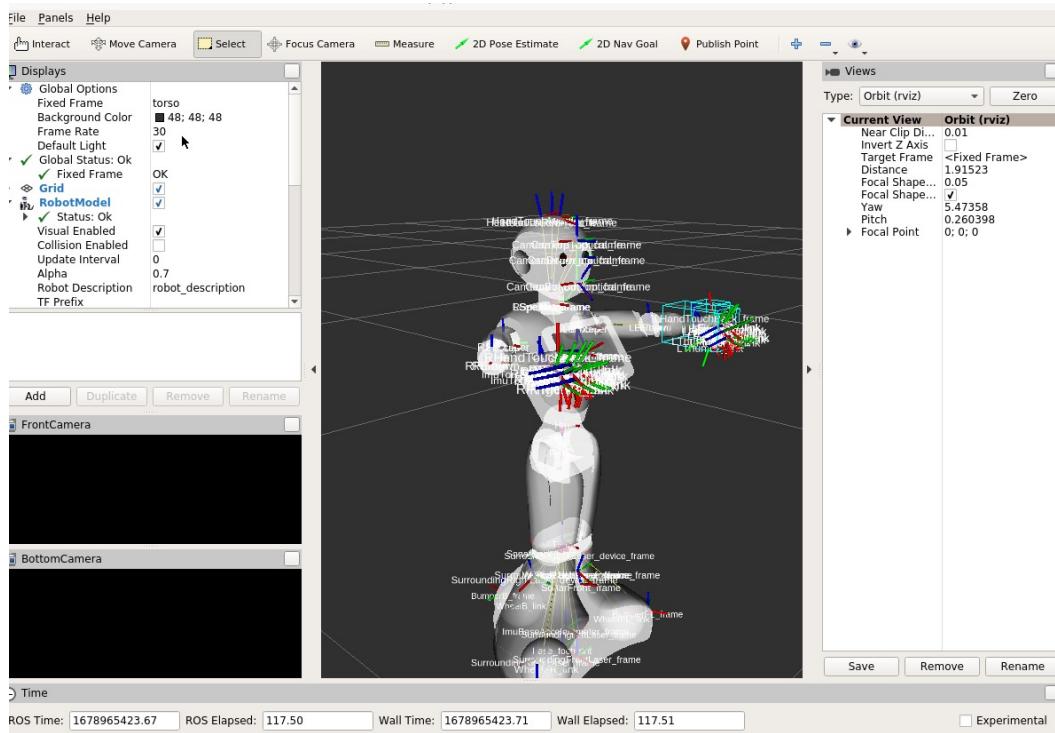


Figura 26: Schermata di RViz con il modello del Pepper

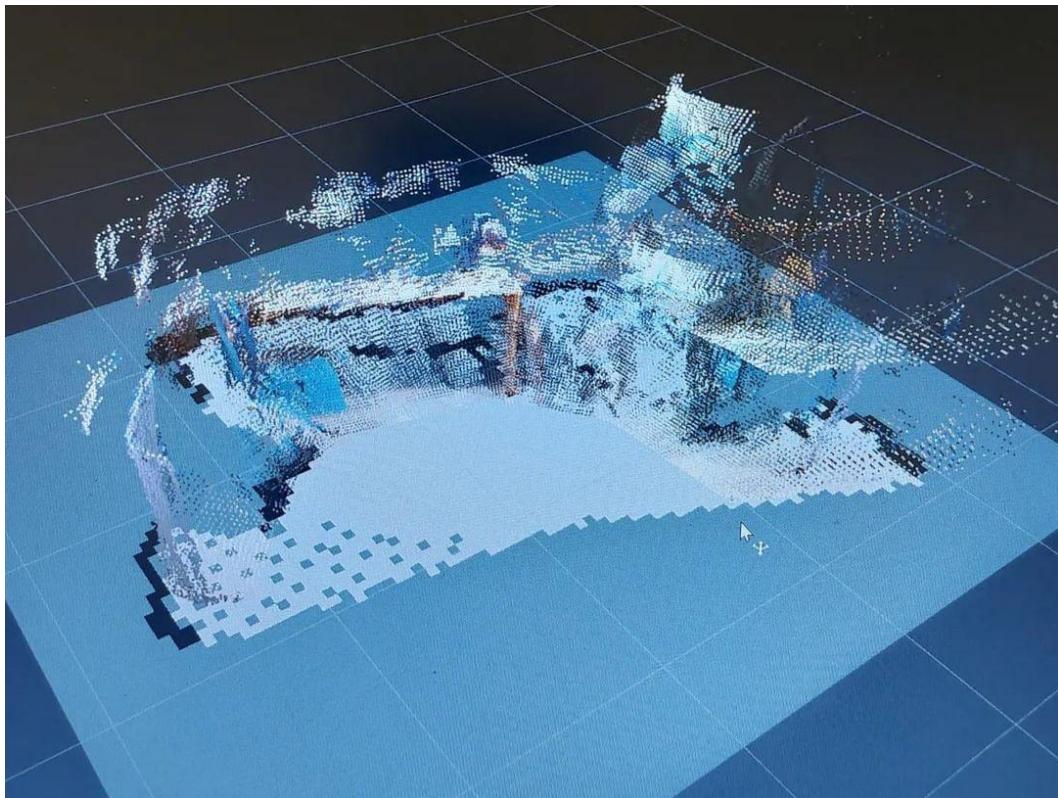


Figura 27: Visualizzazione di una mappa ottenuta con RTabMap

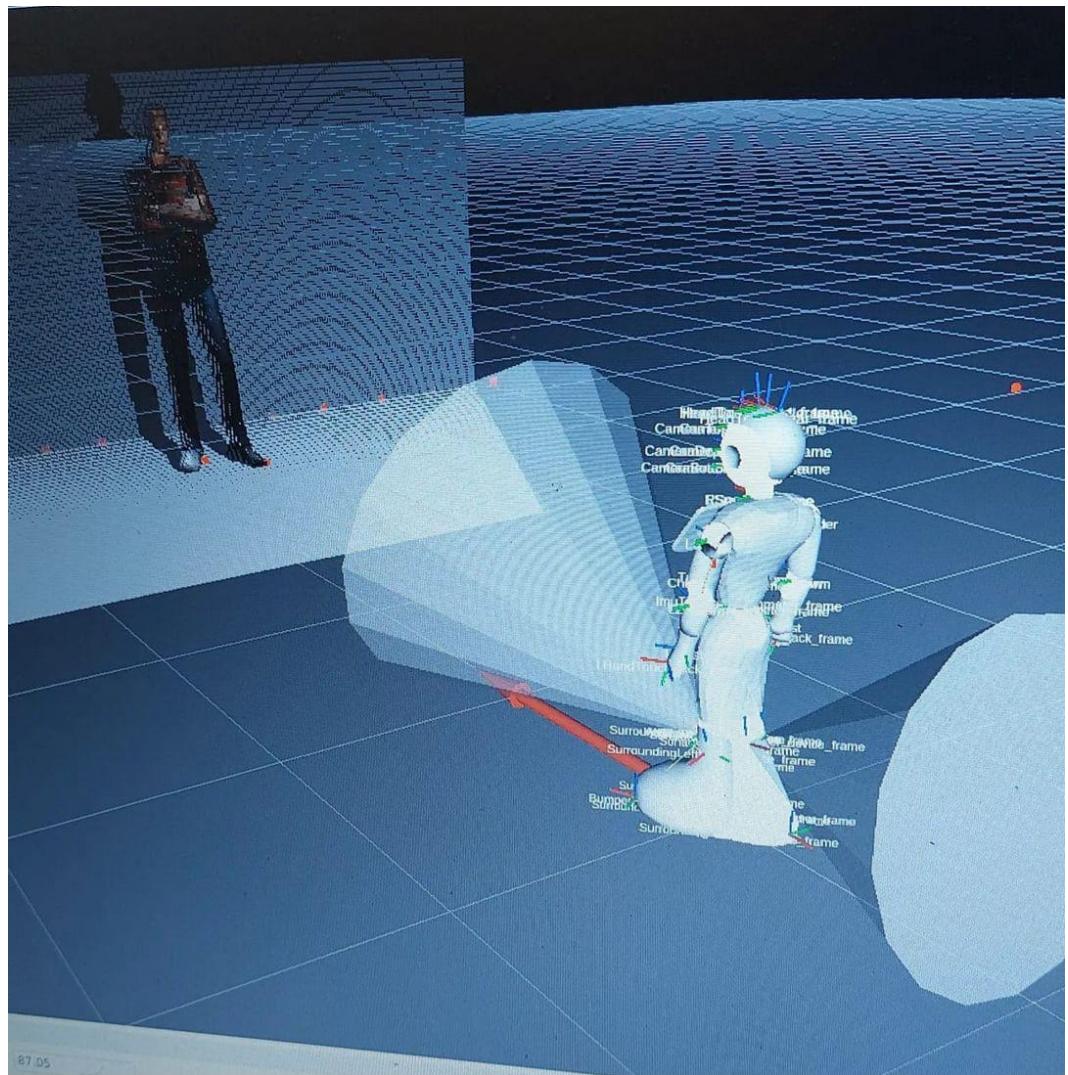


Figura 28: Visualizzazione dei Sonar

5 Evoluzione dell'architettura: da ROS1 a ROS2

Rispetto alla prima fase dei lavori svolti si è deciso di effettuare una transizione da ROS1 a ROS2 per usufruire di importanti migliorie che sono state sviluppate dalla comunità della **Open Robotics Foundation (ORF)**. Per una completa comprensione delle differenze tra i due framework, è consigliata la visione della bibliografia dove è possibile trovare riferimenti alla descrizione dell'architettura della seconda iterazione del sistema [4] e alla documentazione ufficiale [5]. Nella prossima sezione forniamo una breve panoramica relativa al nostro caso d'uso.

- **L'Architettura:** il nucleo di ROS, in entrambe le versioni, è costituito dal suo *middleware*. ROS1 usa un sistema *master-slave* e un protocollo di scambio di informazioni basato su *code di messaggi*. ROS2 offre un **Data Distribution Service (DDS)**, progettato per fornire alta efficienza, robustezza, bassa latenza e scalabilità, contemporaneamente ad un sistema configurabile di **Quality of Service (QoS)** per la modifica di alcuni parametri del sistema (spesso a runtime). Il DDS consente inoltre di supportare in modo migliore sistemi real-time e aiuta a rimuovere il problema del **Single-Point-of-Failure**¹
- **Linguaggi di Sviluppo:** i principali linguaggi di programmazione supportati restano *C++* e *Python* ma nelle loro versioni moderne (standard *C++11* e *Python3.5* come versioni minime) consentendo di avere accesso a nuove feature e librerie oltre che al supporto sul lato della sicurezza e della stabilità del software al costo隐式的 di una non retrocompatibilità con il software sviluppato per ROS1. Inoltre, l'API al centro del framework è stata rivista al fine di poter aggiungere nuove funzionalità (sia da parte del team di sviluppo che dalla community) con più facilità.
- **Librerie Aggiornate:** l'aggiornamento delle tecnologie usate (a partire dai linguaggi e dal DDS) ha consentito di risolvere bug e apportare migliorie a

¹In un sistema informatico un Single Point of Failure (SPOF), letteralmente singolo punto di vulnerabilità, è una parte del sistema, hardware o software, il cui malfunzionamento può portare ad anomalie o addirittura alla cessazione del servizio da parte del sistema. - Wikipedia.

librerie ampiamente utilizzate come lo *stack di navigazione* offerto da **NAV2**.

5.1 Stato dello Sviluppo in ROS2

Passando a ROS2 sono state perdute alcune funzionalità già implementate dagli sviluppatori che hanno curato l'integrazione di *NaoQi - SDK* e ROS. Ad esempio delle interfacce di comunicazione tra ROS e la possibilità di controllare il robot attraverso un joystick, che potrebbero comunque essere implementate in un secondo momento. Ad ogni modo, altre funzionalità sono state aggiunte, come un'applicazione in *android* che è stata installata nel tablet presente sul petto del robot, consentendo agli utenti di utilizzare il robot attraverso un'interfaccia di facile utilizzo.

Durante il periodo di ricerca è stato preferito concentrarsi sullo sviluppo e l'integrazione di quelle funzionalità principali al fine di avviare la connessione con il robot, utilizzare l'SDK attraverso l'avvio di script tramite la libreria *subprocess* di *python*. In particolare, il nucleo del progetto è il pacchetto *Naoqi-Driver2*, con il quale è possibile avviare la connessione con il robot ed esporre nella rete del *DDS* i dati esposti dal robot. A questo vengono agganciati i pacchetti necessari allo svolgimento dei task di SLAM e di navigazione.

Sono stati apportati alcuni fix a tale pacchetto, come l'adattamento del *launchfile* alle specifiche della versione di ROS2 utilizzata (*Foxy*), oppure l'aggiustamento dello stesso per includere l'avvio del visualizzatore *RViz* al fine di permettere allo sviluppatore di capire l'esatto comportamento del robot.

Per l'utilizzo del driver si rimanda al manuale in calce alla relazione.

5.2 Principali Funzionalità Perdute

Tra i pacchetti che il team di sviluppo non ha implementato in ROS2 e, allo stato attuale (10 ottobre 2023), non sembrano in previsione di essere implementate, la più utile è fornita dal pacchetto *naoqi-bridge* che raccoglie i seguenti pacchetti:

- **naoqi-driver-py**: nodi e classi di base utili all’interfacciamento tra ROS e NaoQi-SDK.
- **naoqi-pose**: pacchetto per la gestione dei giunti che governano la posa del robot, permettendo di impostare le pose di default oppure delle pose definite dallo sviluppatore.
- **naoqi-navigation**: Tools per la visualizzazione delle feature di navigazione esposte dai servizi di *ALNavigation* e *ALRecharge*.
- **naoqi-sensors-py**: nodi per l’acquisizione della telecamera, del microfono e dei sonar.
- **naoqi-tools**: tools per generare e modificare i modelli dei robot della Aldebaran, sia tramite i *description model* (ovvero i file *URDF*) e i *visual model* (ovvero modelli e mesh editabili con software di modellazione come *Blender*)

A seguire,abbiamo due pacchetti più utili per l’utilizzo general-purpose:

- **pepper-virtual**: implementa funzionalità di virtualizzazione, al fine di effettuare testing con robot simulati nell’ambiente virtuale di **Gazebo**.
- **pepper-moveit-config**: file di configurazione per il pacchetto **MoveIt!** che consente di muovere i giunti del robot tramite un supporto visuale ma non proprio user-friendly. Uno screen è visibile in fig. 29
- **naoqi-dcm-driver**: interfaccia hardware per il controllo dei robot della Aldebaran tramite ROS, tramite la comunicazione con le librerie proprietarie dell’azienda (*libqi*, *libqicore*). La differenza tra questo driver e gli altri è dovuto al fatto che il controllo è sviluppato più a basso livello e può essere caricato anche direttamente sul robot a patto di compilarlo sulla *virtual machine* proprietaria dell’azienda.²

²attualmente non reperibile causa link della documentazione ufficiale non funzionanti.

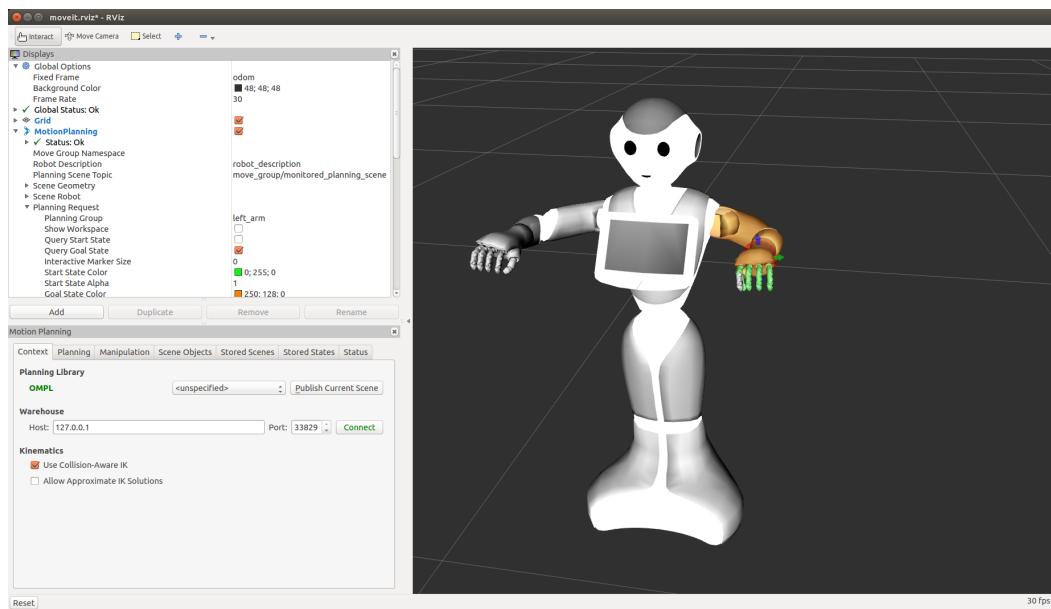


Figura 29: Schermata del pacchetto MoveIt con il file di configurazione per il Pepper

6 SLAM

Il Simultaneous Localization and Mapping (SLAM) è un problema fondamentale nell'ambito della robotica e dell'elaborazione delle immagini. Riguarda la capacità di un robot o di un sistema autonomo di costruire una mappa del suo ambiente circostante mentre determina simultaneamente la sua posizione all'interno di questa mappa. In sostanza, SLAM consente a un robot di *"capire"* dove si trova e di *"imparare"* la disposizione degli oggetti nell'ambiente mentre si muove attraverso di esso, senza disporre di una mappa preesistente.

6.1 Fondamentali ed Approcci

Le fasi fondamentali mostrate nel diagramma di fig. 30 possiamo riassumerle nel seguente modo:

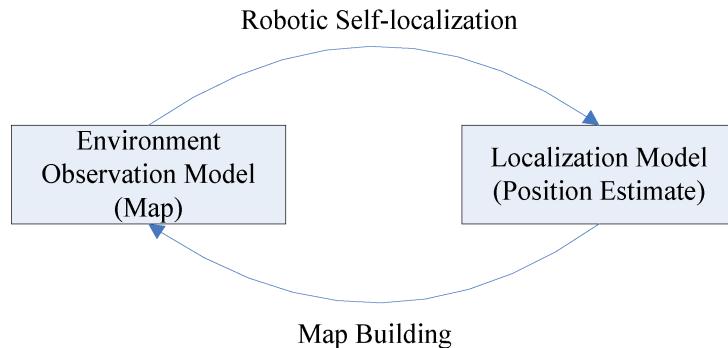


Figura 30: Schema Logico per lo SLAM

- **Mapping:** riguarda la creazione e l'aggiornamento dinamico di una mappa dell'ambiente circostante. La mappa può essere rappresentata in diversi formati, come una mappa topologica, una mappa basata su griglia o una nuvola di punti, a seconda delle esigenze del sistema. L'obiettivo è catturare in modo accurato la geometria e le caratteristiche dell'ambiente.
- **Localization:** si riferisce alla capacità del robot di determinare la sua posizione all'interno della mappa creata. Questo processo coinvolge l'uso di sensori, come telecamere, lidar o odometria, per stimare la posizione e l'orientamento del robot rispetto alla mappa.

Ci sono diverse metodologie ed algoritmi utilizzati per risolvere questo problema, alcuni dei principali sono:

- **Visual SLAM**: utilizza dati provenienti da telecamere per costruire una mappa e localizzare il robot tramite il riconoscimento di feature identificate nelle immagini che vengono registrate.
- **LIDAR SLAM**: fa affidamento su dati provenienti da un sensore **lidar** (*Light Detection and Ranging*) per misurare la distanza tra il robot e gli oggetti circostanti, ricostruendo una mappa simile ad una planimetria. Può essere 2D o 3D a seconda del tipo di scansioni effettuate dal sensore.
- **Hybrid SLAM**: versione ibrida che tramite algoritmi di fusione sensoriale unisce le tipologie precedenti al fine di ottenere una mappa più precisa ed affidabile.

In ogni tipologia di slam, a seconda dell'algoritmo di ricostruzione della posa e della mappa, può essere integrata l'odometria del robot che fornisce una misurazione della velocità e della posizione del robot. Ci sono diversi modi per generare dati di odometria e dipendono principalmente dall'hardware che si ha a disposizione. Tipicamente, si utilizzano gli encoder posti nelle ruote per mappare la velocità di rotazione nelle misure necessarie.

Nel nostro caso d'uso, l'algoritmo di SLAM utilizzato è progettato per essere altamente modulare ed efficiente dal punto di vista della memoria e della gestione dei sensori. Questo ha permesso di sopperire alla scarsa qualità dei laser montati sul pepper utilizzando la telecamera di profondità posta dietro il suo occhio destro. Nonostante i risultati non siano entusiasmanti, si sono dimostrati sufficienti per effettuare operazioni di navigazione.

6.2 Real Time Appearance-Based Mapping

In questa sezione si vogliono fornire dettagli sull'algoritmo utilizzato, al fine di comprenderne meglio caratteristiche e criticità per gli sviluppi futuri.

RTAB-Map[6] è una libreria *open-source* multiplataforma per il Simultaneous Localization and Mapping in tempo reale, progettata per robot mobili e basata su grafi. Questa libreria è sviluppata in C++ e Python ed è nota per affrontare con successo alcune delle sfide tipiche dei tradizionali algoritmi di mappatura e localizzazione, come G-Mapping e ORB-SLAM2.

Le problematiche affrontate da RTAB-Map includono:

- **Difficoltà nell'acquisizione di informazioni dai sensori:** Mappare e localizzare in ambienti complessi può essere complicato a causa della difficoltà nel ricavare informazioni precise dai sensori del robot per la creazione della mappa e la localizzazione.
- **Gestione efficiente della memoria:** L'uso efficiente della memoria è cruciale quando il robot esplora aree più ampie, poiché la quantità di dati da elaborare può crescere notevolmente.
- **Problemi di errore nell'odometria:** Spesso, l'odometria del robot può generare errori, portando a una posizione effettiva del robot che si discosta dalla posizione stimata e questo errore non può far altro che crescere nel tempo.

Per risolvere queste sfide, RTAB-Map è progettata in modo da separare in modo efficace le informazioni provenienti dai sensori e l'odometria dal processo di mappatura e localizzazione, consentendo al robot di gestire meglio queste problematiche. Inoltre, fa un uso intelligente delle informazioni visive per migliorare la precisione, rendendola una soluzione versatile per applicazioni di mappatura e localizzazione in tempo reale per robot mobili. In fig. 31 troviamo un diagramma del nucleo del funzionamento della libreria: Grazie alla sua struttura altamente modulare, RTAB-Map può essere impiegato in diverse configurazioni, in funzione della disponibilità dell'hardware (es: si possiede solo una telecamera ma non un lidar), coinvolgendo sensori come telecamere di profondità, laser e odometria. In situazioni in cui l'odometria delle ruote non è disponibile, RTAB-Map è in grado di recuperarla attraverso l'odometria visuale o tramite il metodo ICP (Iterative Closest Point). Una volta

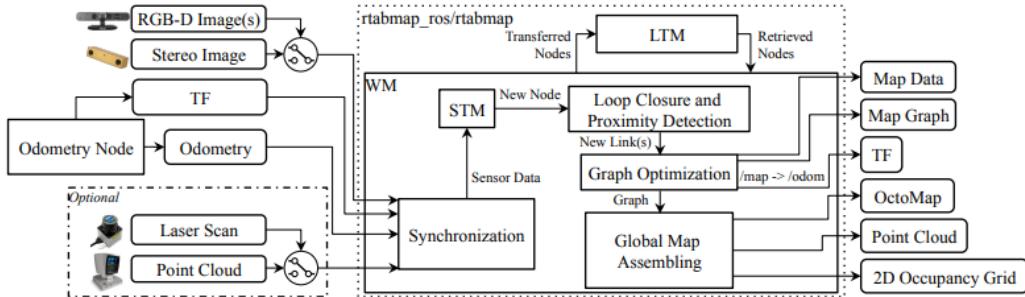


Figura 31: Struttura di RTAB-Map

ottenuti i dati, vengono resi disponibili diversi risultati:

- **Map Data (Dati della mappa):** Questi rappresentano i dati dell'ultima mappa generata dai sensori.
- **Map Graph (Grafo della mappa):** Il grafo della mappa è una struttura dati che contiene tutte le informazioni necessarie per la ricostruzione della mappa. È costituito da nodi e archi (link). I nodi rappresentano le pose del robot e le informazioni ad esse associate, mentre gli archi rappresentano le trasformazioni tra le pose. L'aggiunta di nuovi nodi al grafo è regolata dalla rilevazione di un loop di chiusura (Loop Closure) che avviene attraverso l'analisi delle immagini catturate dalla camera RGB e dei dati laser.
- **Trasformata dalla mappa all'odometria (tf):** Questa informazione rappresenta la trasformazione tra la mappa e l'odometria, consentendo al sistema di allineare correttamente i dati.
- **2D Occupancy Grid (Griglia di occupazione 2D):** Questa rappresenta una mappa utilizzabile per lo stack di navigazione, utile per la navigazione del robot in due dimensioni (ad esempio tramite *Nav2*).
- **PointCloud (Nuvola di punti):** Rappresenta l'insieme dei punti che costituiscono gli ostacoli nell'ambiente.
- **OctoMap (Mappa tridimensionale):** Questa rappresenta una mappa tridimensionale dell'ambiente esplorato.

Il grafo della mappa (Map Graph) è la chiave della ricostruzione della mappa, con nodi che rappresentano le posizioni del robot e archi che rappresentano le trasformazioni tra di esse. L'aggiornamento del grafo avviene attraverso la rilevazione di loop di chiusura o di vicinanza tra nodi. L'aggiunta di un nuovo nodo al grafo viene regolata dal rilevamento di un loop di chiusura (Loop Closure) che avviene tramite l'analisi delle immagini catturate dalla camera RGB e dei valori del laser che verranno utilizzate per aggiornare tutto il grafo.

Per la gestione della memoria, si utilizza una struttura di memoria divisa in due moduli, la *Working Memory* (**WM**) e la *Long Time Memory* (**LTM**), più uno di comunicazione *Short-Term Memory* (**STM**). La WM è una memoria a breve termine contenente informazioni necessarie per la ricostruzione della mappa, mentre la LTM è una memoria a lungo termine che conserva tutte le informazioni raccolte durante l'esplorazione dell'ambiente. Queste due entità comunicano tra loro e con il modulo STM, responsabile di collezionare tutti gli input e ripubblicarli ad una frequenza *user-defined* per la costruzione e l'ottimizzazione del grafo durante la creazione di un nuovo nodo. I nodi vengono valutati e ponderati in base alla loro importanza nell'aggiornamento dello spazio esplorato

Una dimostrazione della mappa ricreata dal pepper è disponibile in fig. 27, dove possiamo vedere un pavimento rappresentante lo spazio noto, in grigio, e gli ostacoli, in nero. I pixel colorati costituiscono il *pointcloud* che l'algoritmo utilizza per generare la mappa.

7 Navigazione Autonoma

La navigazione autonoma in robotica è un problema composto da due fasi chiave: la localizzazione e la pianificazione del percorso. La localizzazione implica la determinazione precisa della posizione del robot all'interno di un ambiente, mentre la pianificazione del percorso riguarda la determinazione del percorso ottimale che il robot deve seguire per raggiungere una destinazione specifica. Per conseguire questi obiettivi, è essenziale che il robot possa percepire l'ambiente circostante attraverso i suoi sensori, rappresentare questa percezione in una mappa e muoversi autonomamente all'interno dell'ambiente.

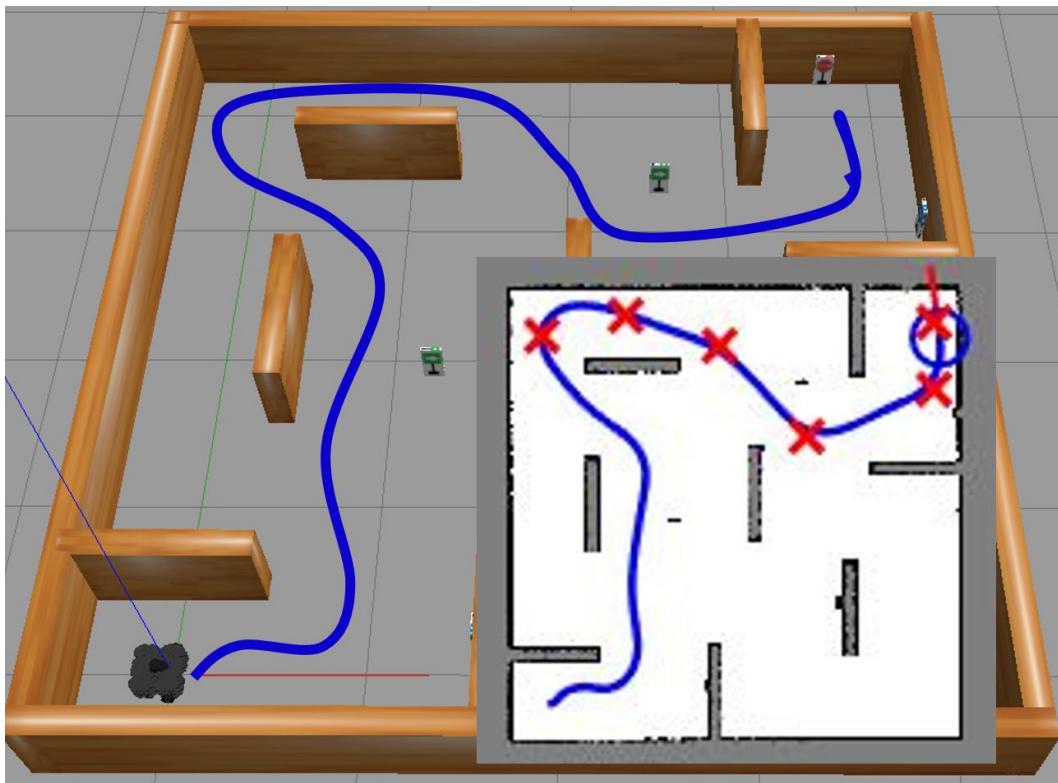


Figura 32: Pianificazione di un percorso (in blu) in un ambiente simulato. Credit: Mathworks

7.1 NAV2

NAV2 è una libreria open-source sviluppata per ROS2 che implementa gran parte delle funzionalità necessarie per affrontare le sfide della navigazione autonoma. In

particolare:

- **Localizzazione del robot:** Nav2 include procedure di localizzazione per determinare con precisione la posizione del robot all'interno di una mappa.
- **Pianificazione del percorso:** Il pacchetto Nav2 offre strumenti per la pianificazione di percorsi ottimizzati, aiutando il robot a determinare la sequenza di azioni da intraprendere per raggiungere i punti di destinazione desiderati.
- **Controllo dei motori:** Nav2 consente un controllo preciso dei motori del robot, garantendo che esso segua con precisione il percorso pianificato.
- **Percezione dell'ambiente:** Nav2 include strumenti per la percezione dell'ambiente circostante, consentendo al robot di rilevare ostacoli e adattare il suo comportamento di navigazione di conseguenza.

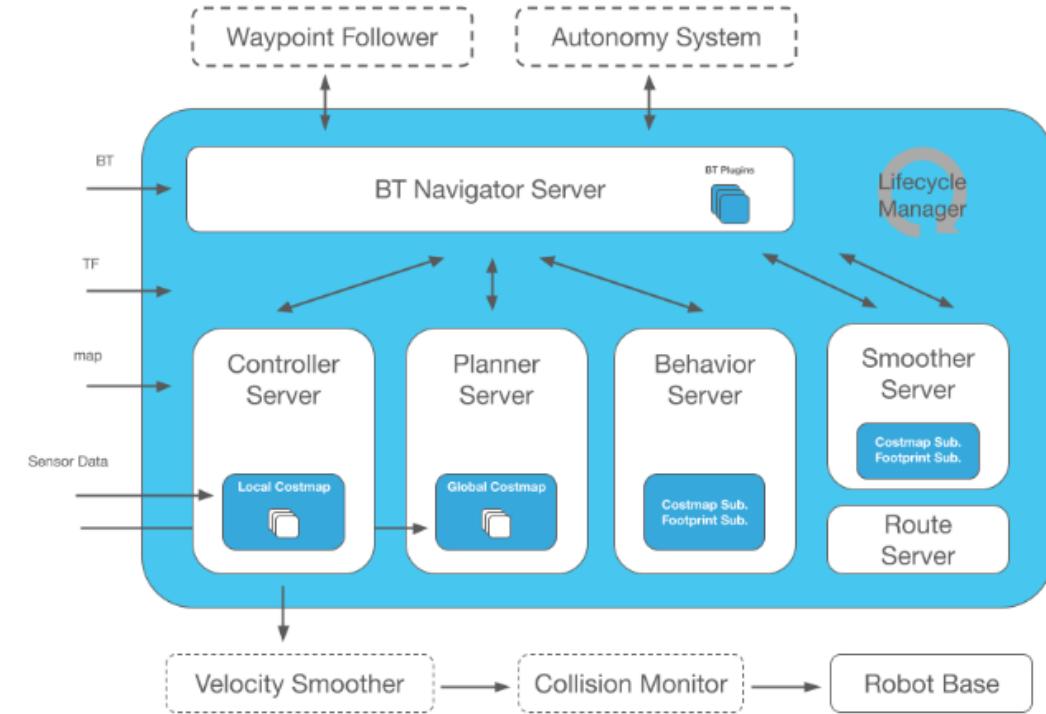


Figura 33: Stack di Navigazione

Grazie alla sua struttura modulare, consente di adattare il comportamento del robot durante la navigazione alle esigenze del caso d'uso in questione, permettendo di tenere conto sia di esigenze specifiche del robot che dell'ambiente che lo circonda.

Tale modularità è ottenuta attraverso una serie di plugin che possono essere attivati e disattivate a runtime in base alle necessità.

7.2 Il Funzionamento

Il funzionamento di questo sistema si basa sull'adozione del paradigma **action-server** di ROS2, in cui un **client** inizia una richiesta al **server** e **segue** il suo progresso attraverso i **feedback** che il server fornisce. La gestione dei numerosi server all'interno di questo gruppo di pacchetti è affidata ai **lifecycle manager**. Questi nodi contengono una **macchina a stati** che gestisce le transizioni tra i diversi stati di navigazione accendendo o spegnendo i server necessari.

I nodi del ciclo di vita contengono transizioni per avviare e spegnere i server ROS 2. Questo contribuisce a garantire un comportamento deterministico dei sistemi ROS durante l'avvio e lo spegnimento. Quando un nodo viene avviato, si trova in uno stato non configurato, in cui elabora solo il costruttore del nodo, che non dovrebbe contenere alcuna configurazione di rete ROS o lettura di parametri. Utilizzando il sistema di lancio o il gestore del ciclo di vita fornito, i nodi devono essere portati allo stato inattivo attraverso una configurazione. Successivamente, è possibile attivare il nodo attraverso la fase di attivazione. Questo stato consente al nodo di elaborare le informazioni e di essere completamente configurato per l'esecuzione. Durante la fase di configurazione, vengono configurati tutti i parametri, le interfacce di rete ROS e, per i sistemi di sicurezza, viene allocata tutta la memoria dinamicamente. La fase di attivazione attiva le interfacce di rete ROS e imposta eventuali stati nel programma per avviare l'elaborazione delle informazioni.

Per lo spegnimento, il nodo passa allo stato di disattivazione, effettua la pulizia, si ferma e poi finisce nello stato finalizzato. Durante queste fasi, le interfacce di rete vengono disattivate e l'elaborazione cessa, la memoria viene deallocata e l'uscita avviene in modo pulito.

Il framework del nodo di ciclo di vita viene ampiamente utilizzato in tutto il progetto e tutti i server ne fanno uso. È considerata la migliore pratica per tutti i sistemi

ROS utilizzare i nodi di ciclo di vita quando possibile.

Il nucleo centrale del sistema di navigazione è costituito da vari componenti fondamentali che operano in sinergia per garantire una navigazione efficace. Questi componenti sono implementati come server che comunicano tra loro utilizzando il paradigma action-server di ROS2:

- **Planner server (Server di Pianificazione):** Questo server è responsabile di calcolare un percorso sulla mappa basandosi su una funzione di costo specifica. Tipicamente, si cerca di trovare il percorso più breve utilizzando algoritmi come Dijkstra, ma è possibile impiegare anche altre tecniche come A* o D* a seconda delle necessità.
- **Controller server (Server di Controllo):** Questo server segue il percorso generato dal Planner server utilizzando informazioni locali sull'ambiente circostante per generare riferimenti adeguati per i motori del robot, garantendo un movimento preciso e sicuro.
- **Recovery server (Server di Recupero):** Il compito di questo server è gestire situazioni di emergenza o comportamenti di recupero quando il robot non riesce a raggiungere un obiettivo o incontra ostacoli. Questo server interviene per affrontare tali situazioni in modo appropriato.
- **Smoother server (Server di Smoothing):** Questo server è incaricato di generare un percorso più fluido a partire dal percorso calcolato dal Planner server. Lo scopo è ridurre le oscillazioni e le rotazioni superflue durante il movimento del robot.
- **Waypoint follower server (Server di Seguimento dei Punti di Passaggio):** Questo server è responsabile di gestire la navigazione quando sono definiti più punti di destinazione, assicurandosi che il robot segua correttamente il percorso desiderato.

Un altro componente importante all'interno dello stack di navigazione offerto da

ROS2 sono gli **behavior trees** (alberi di comportamento). Questi sono strumenti basati su strutture ad albero che gestiscono le attività da svolgere. In particolare, forniscono una logica tramite funzioni base che consentono di creare alberi di comportamento per orchestrare le azioni dei robot in modo flessibile.

7.3 Mappatura e Rappresentazione dell'Ambiente

La sezione relativa alla mappatura e alla rappresentazione dell'ambiente è gestita da due componenti principali:

- **Algoritmi di SLAM:** Questi algoritmi permettono al robot di costruire una mappa dell'ambiente circostante mentre si localizza al suo interno. Nel nostro caso d'uso, il pacchetto RTAB-Map implementa l'algoritmo di SLAM visuale utilizzando immagini catturate dalla telecamera RGB, dati laser e informazioni sull'odometria del Pepper per creare una mappa dell'ambiente.
- **Mappe di costo:** Questa è una modalità per rappresentare l'ambiente circostante attraverso una griglia bidimensionale, dove le celle vengono riempite a seconda della presenza o dell'assenza di ostacoli. Questa rappresentazione è utile per la navigazione poiché fornisce una mappa dettagliata dell'ambiente che il robot può utilizzare per prendere decisioni sulla sua traiettoria.

Tipicamente si distinguono due tipi di mappe di costo: la mappa globale (*global cost-map*) e la mappa locale (*local costmap*). La prima è necessaria per la pianificazione del percorso globale da seguire, mentre la seconda è necessaria per la navigazione locale del robot. Su ognuna di esse è possibile aggiungere delle maschere utili a implementare i comportamenti di navigazione. Ogni maschera viene chiamata **layer** e quelle definite di default si distinguono in:

- **Static Layer:** è il livello piú basso che si aggiunge ad una mappa poiché rappresenta una copia intatta della mappa generata dallo SLAM.
- **Obstacle Layer:** ha il compito di tenere traccia degli ostacoli tramite una lettura dei sensori del robot: le fonti che ROS2 mette a disposizione sono il

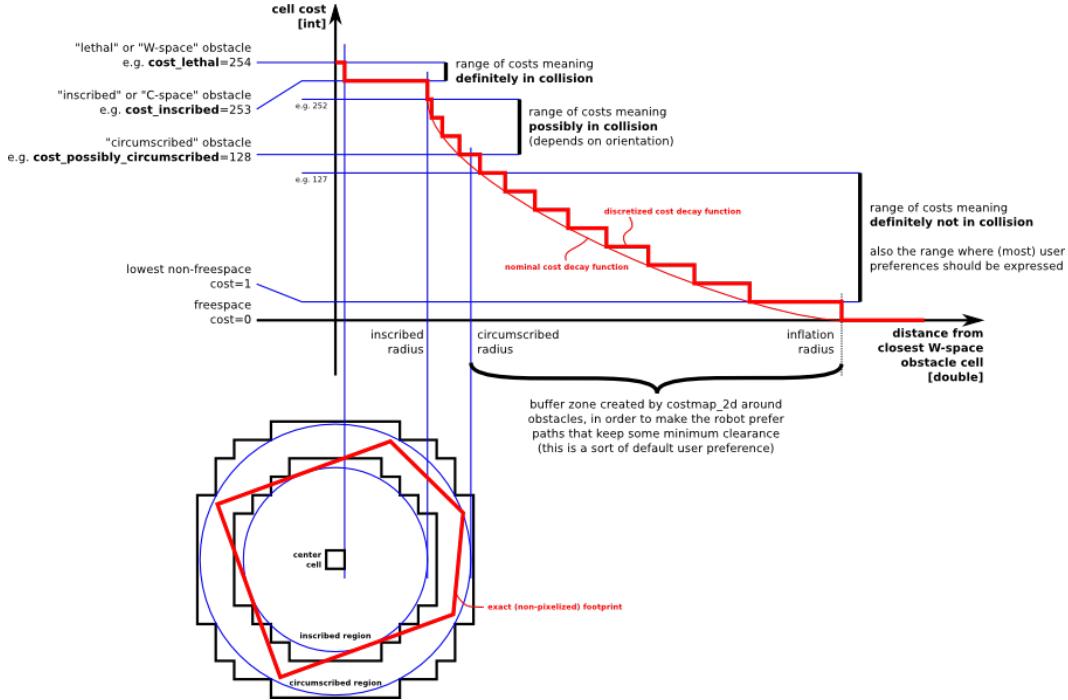


Figura 34: Dettagli sul funzionamento delle costmap

LaserScan con la lettura dei valori dei laser, e il PointCloud2 con la lettura dei valori di punti tre dimensionali. Ad ogni sensore viene definito un livello di importanza basato sulla possibilità che la lettura del sensore possa definire, *marked*, e possa cancellare la presenza di un ostacolo, *cleared*.

- **Inflation Layer:** Ottimizza il lavoro svolto dall’Obstacle Layer. propaga il costo definito da obstacle layer in funzione della distanza dagli ostacoli da -1 , per le celle non note, a 254 per gli ostacoli. In questo modo, si evita che il robot si avvicini troppo agli ostacoli e che possa rimanere bloccato. In fig. 34 è possibile vedere (direttamente dalla documentazione del pacchetto) il significato dei vari componenti di questo strato. Senza scendere nel dettaglio, ogni porzione di mappa viene etichettata nel modo seguente:

- *Lethal*: si riferisce al costo posizionato sull’ostacolo. Se il centroide del robot dovesse trovarsi in questa posizione avverrebbe con certezza una collisione;
- *Inscribed*: si riferisce al costo di una cella che si trova in una distanza

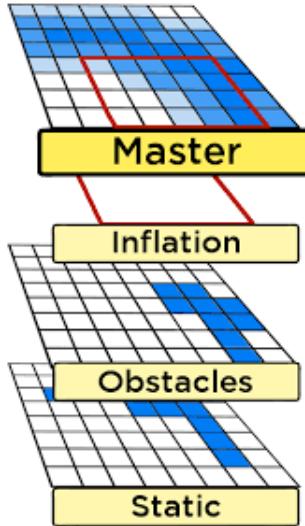


Figura 35: Struttura schematica di una costmap

all'interno del raggio inscritto del robot sul quale si avrebbe una collisione;

- *Possible circumscribed*: si riferisce al costo di una cella che si trova ad una distanza prossima al raggio del robot. Queste celle sono considerati invalicabili o meno a seconda dell'orientamento con cui il robot si avvicina ad esse;
- *Freespace*: si riferisce al costo nullo per il quale non vi sono ostacoli e le celle possono essere attraversate con qualsiasi posa;
- *Unknown*: si riferisce al costo delle celle non ancora esplorate.

- **Voxel Layer:** è un livello che permette di rappresentare l'ambiente circostante in maniera tridimensionale effettuando le stesse operazioni che coinvolgono l'*obstacle layer*.

Questi sono i livelli più utilizzati che vengono aggiunti alle mappe di costo, ma è possibile, importando le opportune librerie, definirne di propri.

7.4 Stato dell'Implementazione

Nel nostro caso, il task di navigazione è stato risolto dopo un'attenta selezione dei parametri di *inflation radius* e di *cost scaling*. Infatti, il laboratorio dove lo sviluppo

è stato portato avanti non era sufficientemente ampio e regolare rispetto alla capacità degli scarsi sensori del robot di offrire misure accurate. Difatti, la navigazione con i parametri di default (fig. 36) è quasi del tutto impossibile in quanto gli ostacoli una volta espansi sono troppo grandi rispetto alla capacità del planner di individuare un percorso. Con la configurazione trovata (fig. 37) la navigazione risulta sempre possibile con lo svantaggio di non avere un *tempo di viaggio* più lungo (il planner va spesso in stato di recovery nel tentativo di cercare un percorso sicuro) e un comportamento più rischioso (il robot tende a passare più vicino agli ostacoli).

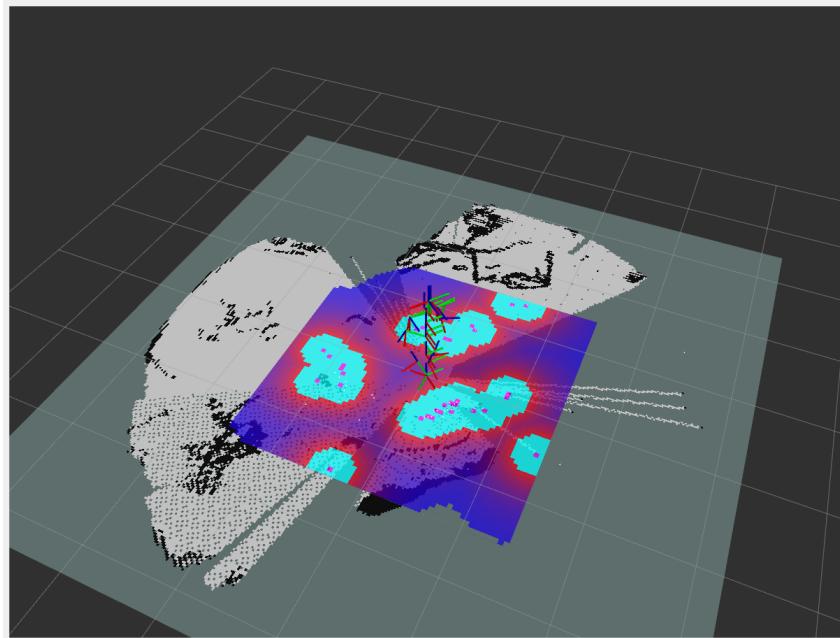


Figura 36: Visualizzazione delle mappe di costo e parametri di default. In rosa gli ostacoli, in celeste le loro espansioni, in blu e in rosso zone della mappa attraversabili con un rischio di incontrare un ostacolo crescente

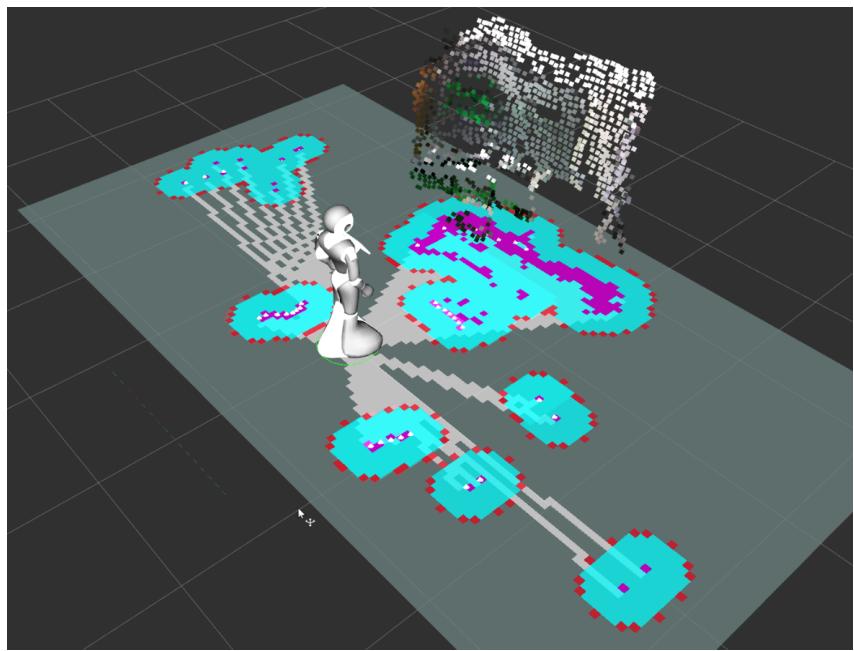


Figura 37: Costmap generata con il robot Pepper e parametri ad-hoc

8 Manuale Utente

In questa sezione cerchiamo di fornire una panoramica sull'utilizzo del software sviluppato al fine di mettere in opera il robot o di portarne avanti lo sviluppo.

8.1 L'Ambiente di Sviluppo

L'ambiente di sviluppo utilizzato e che viene consegnato è basato su un container Docker avviabile tramite opportune estensioni sull'IDE (*Integrated Developer Environment*) gratuito ed open-source di Microsoft, **Visual Studio Code**.

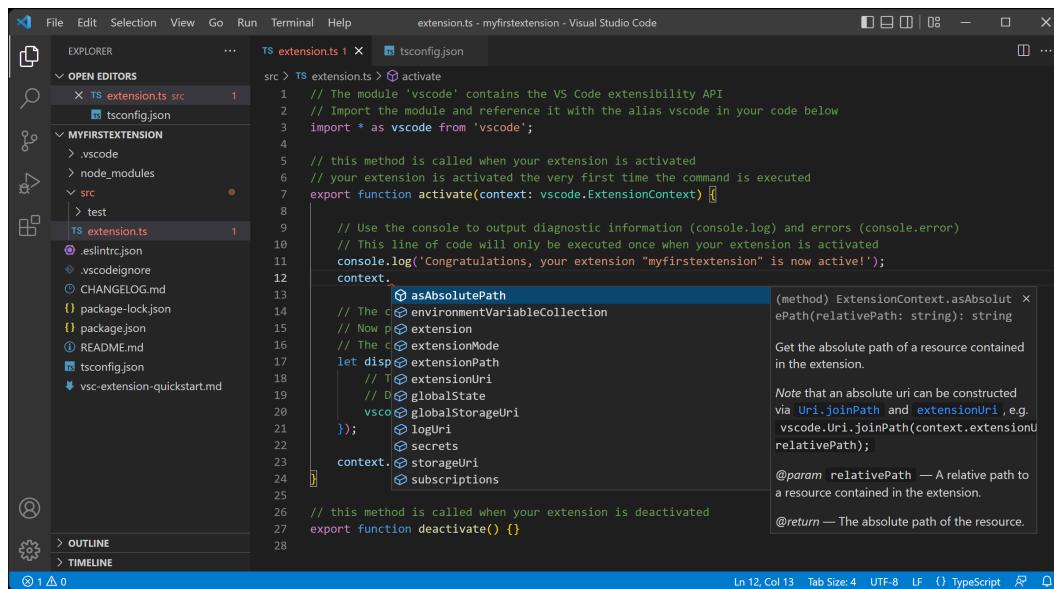


Figura 38: Screenshot dell'IDE

Al fine di avviare il container, è sufficiente aprire la cartella contenente tutti i file necessari tramite lo shortcut da tastiera **ctrl+shift+P** di VS Code e digitare nella barra di ricerca **> open folder in container**. Una volta selezionata la cartella, l'estensione *Remote Container* (installabile tramite la sezione dedicata) eseguirà una scansione dei file alla ricerca dei file di configurazione di Docker. A quel punto si potrà scegliere di avviare il container desiderato, che dovrà prima essere *costruito* nel caso sia la prima volta che viene avviato.

NOTA: Per avviare un container in grado di accedere all'accelerazione grafica offerta da una scheda video **NVIDIA** è necessario installare i relativi driver e il

pacchetto `nvidia-container-toolkit` e configurare il sistema in base alle proprie esigenze. In questo manuale verrà assunto che tali passaggi siano già stati eseguiti e non verrà fornito supporto per eventuali problemi. Per maggiori informazioni si rimanda alla [Documentazione Ufficiale](#)

NOTA: Se si ha a disposizione una GPU Intel/AMD non è necessario eseguire alcun passaggio aggiuntivo oltre all'installazione dei rispettivi driver.

8.2 Struttura del Container

Il container è suddiviso in due *workspace*. Il primo è quello che è effettivamente presente nel filesystem della macchina *host*, mentre il secondo è presente nel volume che viene creato da Docker al momento della creazione del container. In particolare, all'atto dell'avvio del container, viene montata la cartella dove risiedono tutti i file nella radice del volume virtuale, alla voce `/workspaces`.

Questa suddivisione è stata scelta al fine di evitare che i file generati alla compilazione dei pacchetti vengano salvati in locale e per evitare di annidare diverse repository di terze parti nella cartella principale. In questo modo, se dovessero esserci problemi, è possibile distruggere e ricreare il container da zero senza perdere ciò che è stato scritto dallo sviluppatore.³

NOTA: per navigare all'interno del file system con VS Code, basta digitare

```
> code <folder-path>
```

 per aprire una sessione dell'IDE nella cartella specificata.

8.3 Script Principali e Flusso di lavoro

Ci sono due *script* principali che sono stati scritti al fine di migliorare la qualità della vita dello sviluppatore.

- `setup.sh`: Questo file è usato per aggiornare/scaricare le dipendenze delle repository di ROS che sono state inserite nel file `ros2.repos`. Va esegui-

³Nel caso si distrugga il container, qualsiasi modifica fatta al suo interno una volta lanciato verrà perduta, quindi preoccuparsi di salvare in locale tutto ciò che si ritiene necessario.

to prima di compilare un nuovo pacchetto, per assicurarsi di avere tutte le dipendenze necessarie.

- **build.sh**: Questo file è usato per compilare il workspace ROS2. Questa dovrebbe essere l'unico modo con cui lo sviluppatore vorrebbe compilare il workspace per evitare che il compilatore di ROS2 (`colcon`) scriva nella *working directory* le cartelle `install`, `build` e `log` e creare molteplici workspace che potrebbero causare conflitti.

L'ideale flusso di lavoro per l'aggiunta di un pacchetto è il seguente:

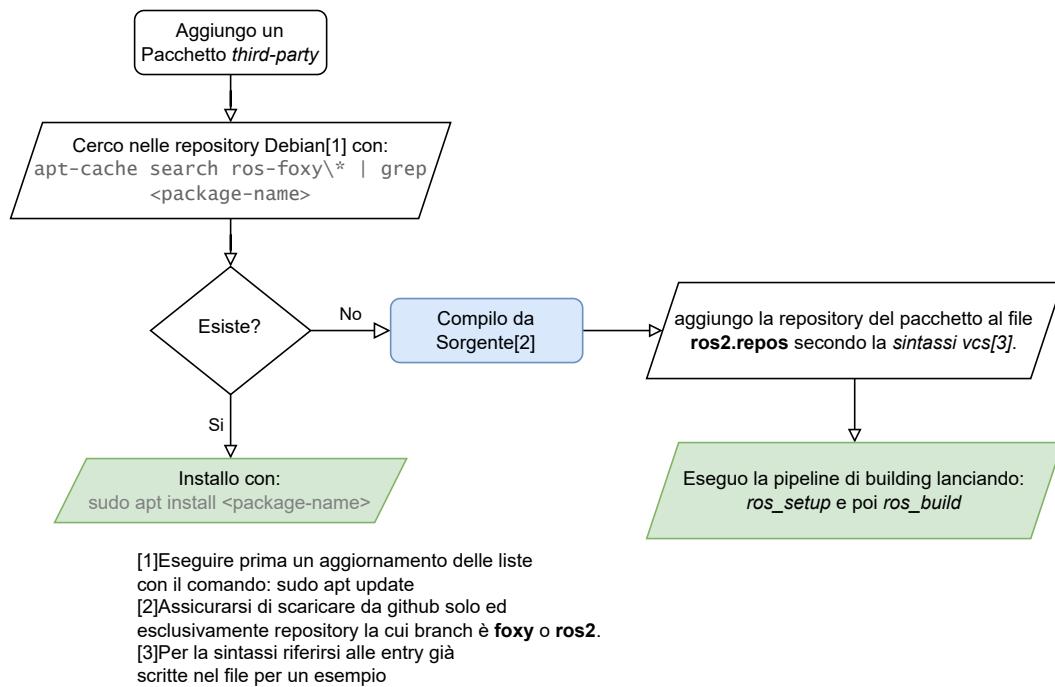


Figura 39: workflow

8.4 Il pacchetto PepperMeta

Il pacchetto `pepper_meta` rappresenta la collezione di funzionalità sviluppate e si costituisce di 4 parti:

- **pepper_nav**: collezione di *launch-files* e parametri di configurazione per eseguire i task di SLAM e Navigazione tramite RTAB-Map e NAV2.

- `ros_naoqi_motion`: pacchetto di interazione tra l'architettura ROS, quella Android presente sul tablet del robot e, **soprattutto**, l'allacciamento alle funzionalità dell'SDK al fine di sviluppare funzionalità di movimento dei giunti.
- `ros_naoqi_tts`: simile al pacchetto precedente, ma per sviluppare le funzionalità di *text-to-speech*.
- `pepper_dashboard`: applicazione android che viene eseguita dal tablet del robot. Offre diverse funzionalità tra cui la possibilità di richiedere al computer host di avviare il servizio *text-to-speech* del robot o di muovere il suo braccio per porgere al paziente il dispositivo per effettuare la misura.

Per quanto riguarda l'utilizzo dell'applicazione Android, non verrà esposto qui il suo funzionamento, ma verrà inserito un diagramma di come avviene la comunicazione tra le entità in gioco: ROS, Android e NAOqi SDK. Prima però vediamo i comandi necessari ad attivare il robot e a lanciare l'esecuzione degli algoritmi di SLAM e di Navigazione, oltre che i server per l'interazione con Android.

NOTA: i requisiti fondamentali per l'esecuzione dei programmi sono gli indirizzi IP delle macchine in questione. In particolare:

- `nao_ip` indirizzo del robot pepper. Ottenibile premendo una volta il bottone sul suo torso, dietro il tablet.
- `host_ip` indirizzo della macchina host. Ottenibile digitando il comando `> hostname -I` su un terminale. L'indirizzo deve essere uno assegnato dallo stesso router al quale è connesso il robot.
- `tablet_ip` indirizzo del tablet del robot. Ottenibile dalle impostazioni del tablet stesso.
- `porte` le porte per i servizi di connessione. Per il Pepper, la porta di default è la `9559` (non è possibile cambiarla in quanto è statica sul robot), per il servizio di movimento è la `9999` mentre per quello di *text-to-speech* `9090`.

Il processo di attivazione e di esecuzione è il seguente⁴:

1. Si avvia il robot tramite il comando

```
> ros2 launch naoqi_driver naoqi_driver.launch.py nao_ip:=<nao_ip>  
nao_port:=<nao_port>
```

Questo scatena il processo di avviamento di tutti i nodi ROS esposti dal driver e (nella versione modificata da noi), porta il robot a svegliarsi e ad assumere la posizione eretta.

2. Si avvia il sistema di slam tramite

```
> ros2 launch pepper_nav rtabmap.launch.py ip_address:=<host_ip>
```

Viene avviato l'algoritmo di SLAM e si apre il server di *Vizanti*, un pacchetto in grado di permettere la visualizzazione della navigazione tramite una pagina web. La porta selezionata per questo servizio è la 5000

3. Si avvia il sistema di navigazione tramite

```
> ros2 launch pepper_nav navigation.launch.py
```

Viene avviato lo stack di navigazione di NAV2, passando i parametri selezionati in modo implicito dal launch-file.

4. Si avvia il server di text-to-speech tramite la combinazione di:

```
> ros2 launch ros_naoqi_tts tts_publisher.launch.py  
host_ip:=<host_ip> host_port:=<host_port> nao_ip:=<nao_ip>  
nao_port:=<nao_port>
```

```
> ros2 launch ros_naoqi_tts tts_subscriber.launch.py  
nao_ip:=<nao_ip> nao_port:=<nao_port>
```

⁴gli argomenti vengono passati con la sintassi nome:=<valore>, dove il valore va specificato senza <>. Se non viene specificato un valore, viene usato quello di default scritto nel launch-files.

In questo modo viene avviato il web-server che tramite protocollo TCP è in grado di scambiare dati tramite socket aperta sulla coppia indirizzo-porta specificata e allo stesso tempo il nodo per effettuare la richiesta del servizio text-to-speech tramite API dell'SDK. Quando dall'applicazione verrà fatta richiesta, verrà inoltrato un messaggio contenente il testo da riprodurre (codificato in *utf-8*) dal tablet alla macchina host e verrà ripubblicato un messaggio contenente quel testo sul topic dove il nodo responsabile di avviare il servizio di text-to-speech è in ascolto.

5. Si avvia il server per il servizio di movimento tramite

```
> ros2 launch ros_naoqi_motion motion.launch.py host_ip:=<host_ip>
    host_port:=<host_port> nao_ip:=<nao_ip> nao_port:=<nao_port>
```

In questo modo si avvia uno stack di comunicazione similare al precedente e vengono avviati gli opportuni nodi responsabili della gestione del movimento del braccio tramite API.

8.5 Dettagli sull'utilizzo di NAOqi SDK tramite ROS

Al fine di poter sfruttare le API proprietarie per eseguire alcune parti dei task è stato scelto di utilizzare la libreria *subprocess* di python. Tramite questa libreria è stato possibile eseguire codice il codice *python2* in cui le API sono state scritte, bypassando il problema di compatibilità. Lo schema adottato è potenzialmente riproducibile per implementare qualsiasi funzionalità originariamente offerta e perduta passando allo sviluppo in *Python3*.

NOTA: dovrebbe comunque essere possibile l'implementazione delle API in modo *nativo* (senza l'esecuzione di sottoprocessi appositi) utilizzando l'SDK scritto in *C++*, ma avendo poche conoscenza di questo linguaggio non abbiamo approfondito la questione. Potrebbero esserci delle incompatibilità dovute alla versione del linguaggio utilizzata e potrebbe essere necessario un lavoro atten-

to per compilare correttamente i pacchetti ROS, che ovviamente dovrebbero essere scritti in *C++*.

Per illustrare lo schema di interazione consideriamo quello adottato nel pacchetto `ros_naoqi_motion`, esposto in fig. 40,

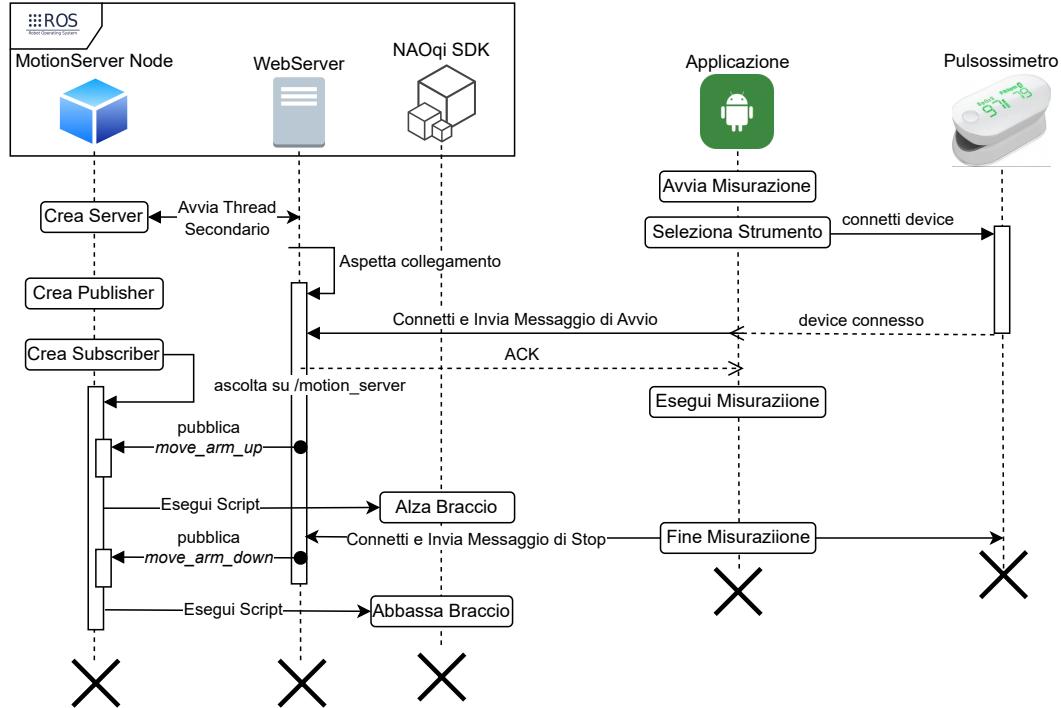


Figura 40: Interazione tra NAOqi e ROS

Una volta avviato il nodo del *MotionServer*, durante la costruzione dell’oggetto, viene creato un oggetto *WebServer* che apre una socket all’indirizzo `host_ip:host_port` (settato di default a `127.0.0.1:9999`) e va in attesa continua di un client che si connetta ad esso. Il server viene avviato lanciando un thread separato, in modo tale da non bloccare l’esecuzione del nodo e permettere l’interazione con l’applicazione in maniera asincrona. Al momento della creazione del nodo, vengono avviati anche un *publisher* ed un *subscriber*, sul topic `/motion_server`. La callback del subscriber, viene attivata quando arriva un messaggio su questo topic.

L’applicazione (che dispone dell’indirizzo ip dell’host e della sua porta), quando viene premuto il bottone per avviare la procedura di misurazione e successivamente quello del pulsossimetro, avvia una connessione sia con il dispositivo che con il

server. Una volta che la connessione con il server viene stabilita, viene inviato un messaggio ad esso inviando una stringa di testo. Quando il server riceve il messaggio, tramite un riferimento al nodo che l'ha creato, pubblica un messaggio sul topic `/motion_server`, svegliando il subscriber.

Quando la callback viene triggerata, viene avviato un sottoprocesso tramite il comando *subprocess*, per avviare uno script scritto in *python2* che si occupa di connettersi al robot tramite API, istanziare tutti i servizi necessari al movimento dei suoi giunti, ed esegue la sequenza di comandi per muovere il suo braccio in alto, porgendolo al paziente (fig. 41).

Quando la misurazione è terminata e viene premuto il pulsante *indietro* sul tablet, l'applicazione invia un nuovo messaggio al server, per avviare una procedura analoga alla precedente, ma chiedendo di abbassare il braccio.

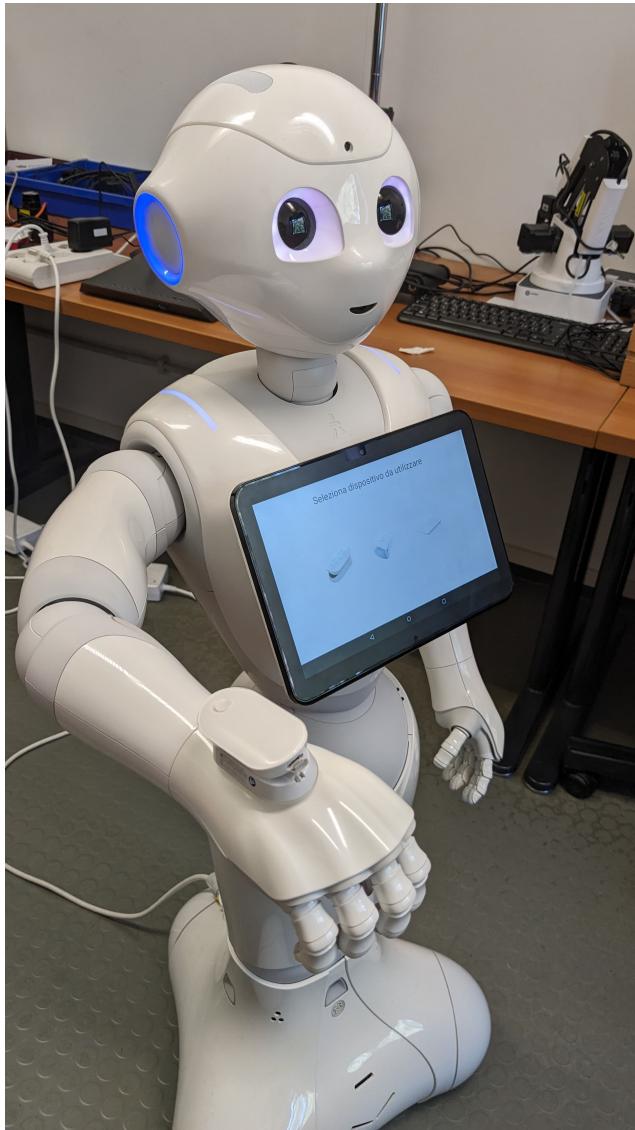


Figura 41: Il pepper con il braccio alzato, che porge il pulsossimetro

9 Sviluppi Futuri

Possibili sviluppi futuri per il progetto dovrebbero riguardare principalmente al miglioramento della stabilità dell'applicazione e all'implementazione delle funzionalità mancanti. In particolare, per quanto riguarda la stabilità, in primo luogo bisognerebbe effettuare un aggiornamento del driver e dei pacchetti necessari al suo funzionamento per renderlo compatibile con le versioni recenti di ROS, dato che la branch da noi in uso (*foxy*) è nella sua fase *end-of-life*.

Per le funzionalità mancanti, si potrebbe prima procedere ad effettuare un *disaccoppiamento* del sistema di connessione alle API NAOqi, per permettere di aggiungere più comodamente nuove funzionalità e possibilmente crearne di più complesse permettendo un approccio riutilizzabile; ad esempio si potrebbe scrivere un nodo in grado di fare da interfaccia di *entrypoint* per tutti i servizi delle API.

Un modo per creare funzionalità più complesse, potrebbe essere quello di riuscire a comunicare con il sottoprocesso che viene avviato a runtime, in modo tale da permettere lo scambio di dati tra ROS e Python2 e generare un flusso d'esecuzione complesso e che permetta di interagire direttamente con le API.

Per quanto riguarda la fase di navigazione e di SLAM, passi avanti potrebbero essere fatti in entrambi i sensi:

- Per quanto riguarda la fase di mappatura e localizzazione, si potrebbero montare sensori migliori sul robot ed aggiungere un *companion-pc* per pilotarli. Ad esempio, una telecamera di profondità dotata di una risoluzione maggiore e nativamente compatibile con ROS, come una *Zed*, che è dotata anche di un sistema LIDAR di molto più potente dei tre laser alla base del pepper.
- Per la navigazione, migliorare il sistema di SLAM sarebbe un ottimo punto di partenza, ma anche un tuning migliore dei parametri dell'*inflation layer* per generare dei gradienti di costo *più smooth* potrebbero migliorare il modo con cui il robot si muove.

Riferimenti bibliografici

- [1] O. R. Foundation, “Ros - robotic operating system.” [Online]. Available: <https://www.ros.org/>
- [2] A. U. R. Group, “Naoqi sdk.” [Online]. Available: http://doc.aldebaran.com/2-5/dev/programming_index.html
- [3] “Ros-naoqi.” [Online]. Available: <https://github.com/ros-naoqi>
- [4] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022. [Online]. Available: <https://www.science.org/doi/abs/10.1126/scirobotics.abm6074>
- [5] O. R. Foundation, “Ros2 - robotic operating system 2,” <https://docs.ros.org/en/foxy/index.html>.
- [6] M. Labbé and F. Michaud, “Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation,” *Journal of Field Robotics*, vol. 36, no. 2, pp. 416–446, 2019.